How to Trust your Peephole Rewrites: Automatically Prove Them For Arbitrary Width!

Siddharth Bhat

UNIVERSITY OF CAMBRIDGE

Extremely Collaborative Work!

- bv_decide was developed by the Lean Focused Research Organization (Henrik Böving implemented bv_decide, supervised by Kim Morrison, Leonardo De Moura)

- Josh Clune implemented the LRAT checker.

- Many folks contributed to the Bitvector Library: Abdalrhman Mohamed, Alex Keizer, Harun Khan , Henrik Böving , Joe Hendrix , Kim Morrison , Leonardo de Moura , Luisa Cicolini, Siddharth Bhat, Tobias Grosser, Wojciech Nawrocki, Joe Hendrix, ...

- Collaboration with Léo Stefanesco to implement arbitrary-width decision procedures.

- Chris Hughes wrote the first version of the decision procedure in Lean.

Alive Is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

Alive Is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

Alive Is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

Alive Is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

```
Transformation seems to be correct!
```

Alive Is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

Alive is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome!

[Inst

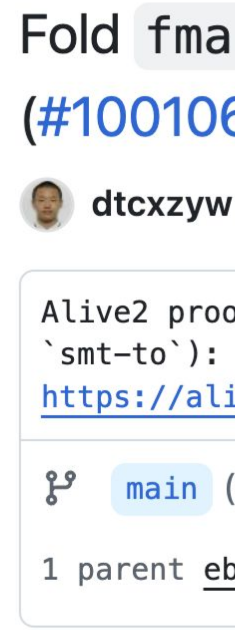
only

xc

Fixes

Alive

1 par



Fold fma

(#10010

dtcxzyw

Alive2 proo

`smt-to`):

https://ali

main (

1 parent eb

[InstCombine] fo

|| (a != c &&

== (b != c) (#

zjaffal authored on

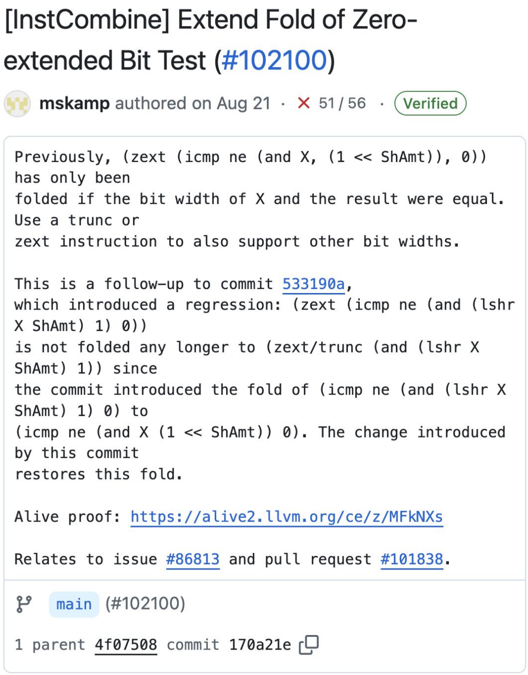
resolves #92966

alive proof

https://alive2.llvm.c

main (#94915)

1 parent 3ae6755 comm



[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e

Alive is Awesome! How Does It Work?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

Alive is Awesome! How Does It Work?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

```
(set-logic QF_UFBV)

(define-fun src
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvudiv x (_ bv32 1)))

(define-fun tgt
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvlshr x (_ bv32 32)))
```

Alive is Awesome! How Does It Work?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}
```

```
(set-logic QF_UFBV)

(define-fun src
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvudiv x (_ bv32 1)))

(define-fun tgt
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvlshr x (_ bv32 32)))
```

"does src equal tgt for all inputs?"

→ 🧞 **Solver** ✅ ❌

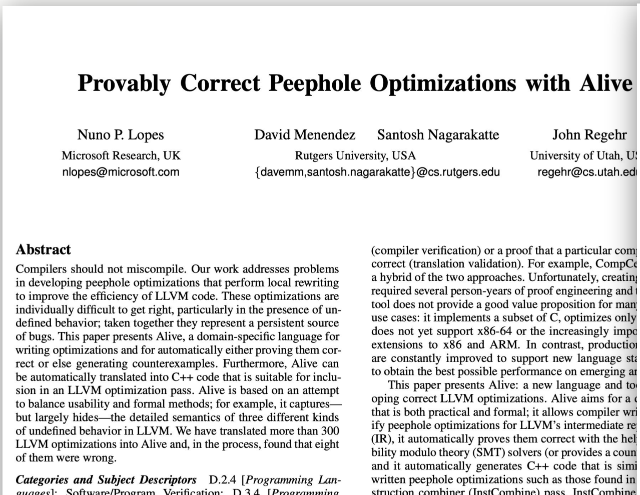
Alive is Awesome! How Does It Work?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret
}
```

```
(set-logic QF_UFBV)

(define-fun src
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvudiv x (_ bv32 8192)))
```

"does src equal tgt for all inputs?"

LLVM $\xrightarrow{\text{Alive}}$ **QF_BV** → 🧞 **Solver** ✅ ❌



Provably Correct Peephole Optimizations with Alive

Nuno P. Lopes
Microsoft Research, UK
nlopes@microsoft.com

David Menendez Santosh Nagarakatte
Rutgers University, USA
{davemm,santosh.nagarakatte}@cs.rutgers.edu

John Regehr
University of Utah, U
regehr@cs.utah.ed

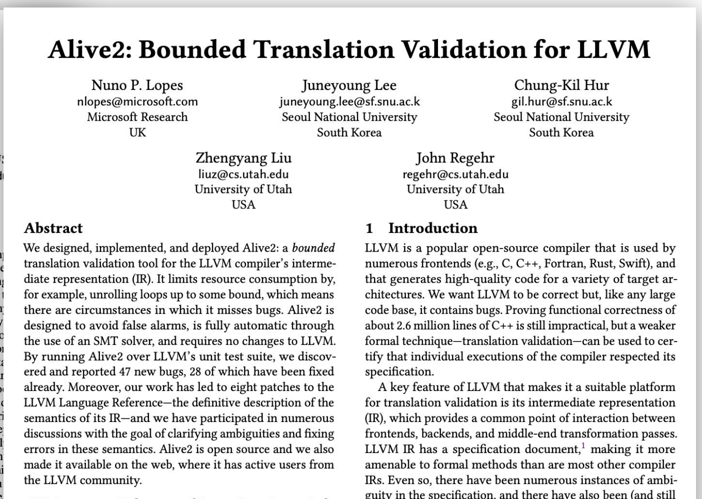
Abstract

Compilers should not miscompile. Our work addresses problems in developing peephole optimizations that perform local rewriting to improve the efficiency of LLVM code. These optimizations are individually difficult to get right, particularly in the presence of undefined behavior; taken together they represent a persistent source of bugs. This paper presents Alive, a domain-specific language for writing optimizations and for automatically either proving them correct or else generating counterexamples. Furthermore, Alive can be automatically translated into C++ code that is suitable for inclusion in an LLVM optimization pass. Alive is based on an attempt to balance usability and formal methods; for example, it captures—but largely hides—the detailed semantics of three different kinds of undefined behavior in LLVM. We have translated more than 300 LLVM optimizations into Alive and, in the process, found that eight of them were wrong.

Categories and Subject Descriptors D.2.4 [*Programming Languages*]: Software/Program Verification; D.3.4 [*Programming*

(compiler verification) or a proof that a particular com
correct (translation validation). For example, CompCe
a hybrid of the two approaches. Unfortunately, creatin
required several person-years of proof engineering and
tool does not provide a good value proposition for man
use cases: it implements a subset of C, optimizes only
does not yet support x86-64 or the increasingly impo
extensions to x86 and ARM. In contrast, production
are constantly improved to support new language sta
to obtain the best possible performance on emerging ar

This paper presents Alive: a new language and to
oping correct LLVM optimizations. Alive aims for a
that is both practical and formal; it allows compiler wri
ify peephole optimizations for LLVM's intermediate re
(IR), it automatically proves them correct with the hel
bility modulo theory (SMT) solvers (or provides a coun
and it automatically generates C++ code that is simi
written peephole optimizations such as those found in
struction combiner (InstCombine) pass. InstCombine



Alive2: Bounded Translation Validation for LLVM

Nuno P. Lopes
nlopes@microsoft.com
Microsoft Research
UK

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University
South Korea

Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University
South Korea

Zhengyang Liu
liuz@cs.utah.edu
University of Utah
USA

John Regehr
regehr@cs.utah.edu
University of Utah
USA

Abstract

We designed, implemented, and deployed Alive2: a *bounded* translation validation tool for the LLVM compiler's intermediate representation (IR). It limits resource consumption by, for example, unrolling loops up to some bound, which means there are circumstances in which it misses bugs. Alive2 is designed to avoid false alarms, is fully automatic through the use of an SMT solver, and requires no changes to LLVM. By running Alive2 over LLVM's unit test suite, we discovered and reported 47 new bugs, 28 of which have been fixed already. Moreover, our work has led to eight patches to the LLVM Language Reference—the definitive description of the semantics of its IR—and we have participated in numerous discussions with the goal of clarifying ambiguities and fixing errors in these semantics. Alive2 is open source and we also made it available on the web, where it has active users from the LLVM community.

1 Introduction

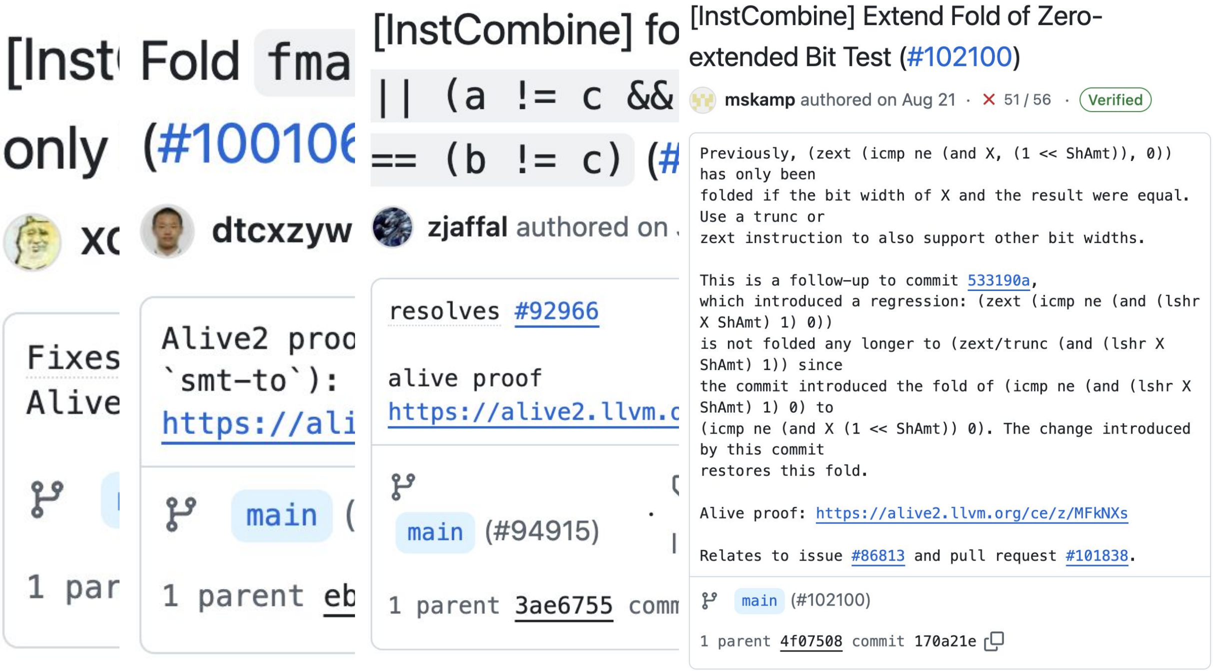
LLVM is a popular open-source compiler that is used by numerous frontends (e.g., C, C++, Fortran, Rust, Swift), and that generates high-quality code for a variety of target architectures. We want LLVM to be correct but, like any large code base, it contains bugs. Proving functional correctness of about 2.6 million lines of C++ is still impractical, but a weaker formal technique—translation validation—can be used to certify that individual executions of the compiler respected its specification.

A key feature of LLVM that makes it a suitable platform for translation validation is its intermediate representation (IR), which provides a common point of interaction between frontends, backends, and middle-end transformation passes. LLVM IR has a specification document,[1] making it more amenable to formal methods than are most other compiler IRs. Even so, there have been numerous instances of ambiguity in the specification, and there have also been (and still

SMT Solver

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome! How Does It Work?



[Inst

xc

Fixes
Alive

1 par

Fold fma only (#10010

dtcxzyw

Alive2 proo
`smt-to`):
https://ali

main (

1 parent eb

[InstCombine] fo
|| (a != c &&
== (b != c) (#

zjaffal authored on

resolves #92966

alive proof
https://alive2.llvm.c

main (#94915)

1 parent 3ae6755 comm

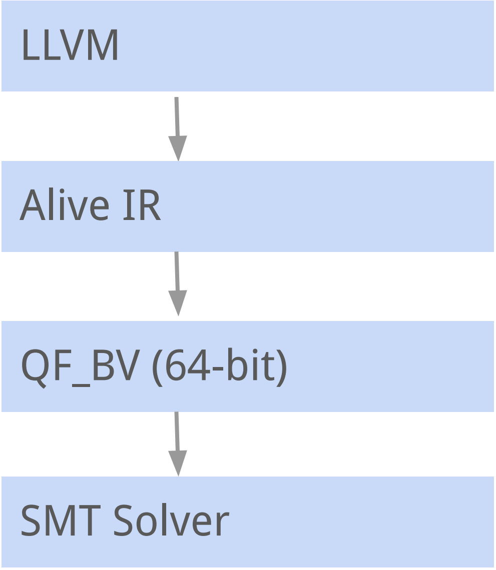
[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

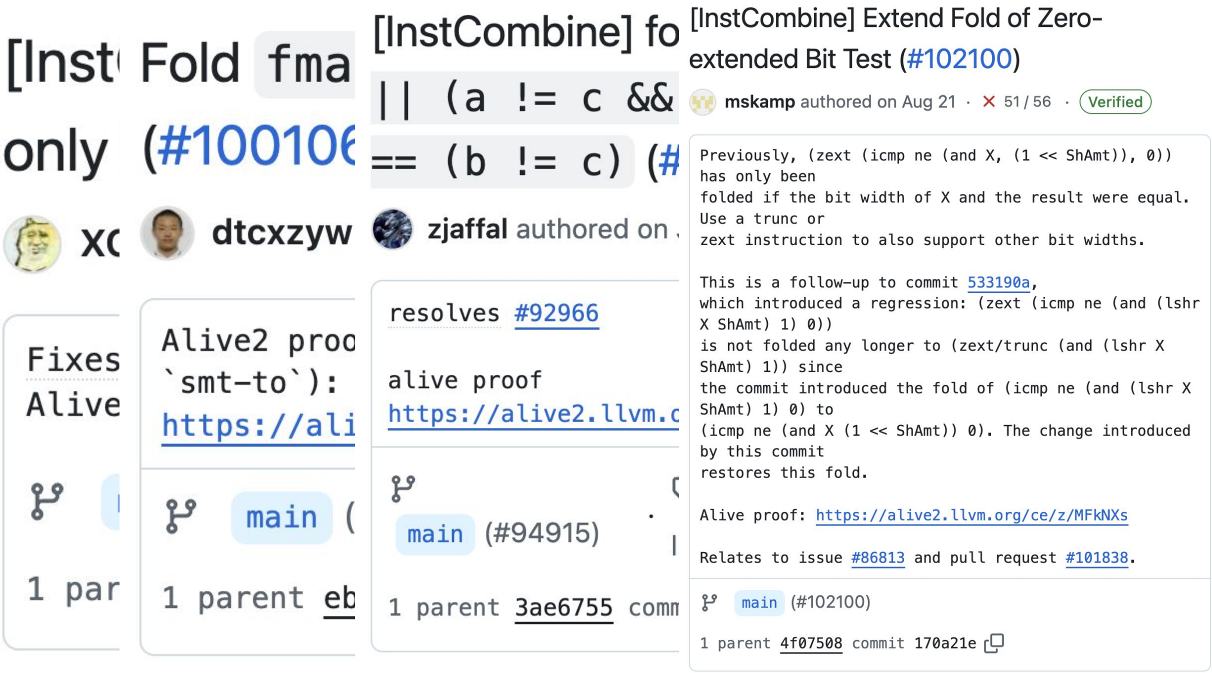
```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e



Alive is Awesome! How Does It Work?



[Inst Fold `fma` only (#10010

xc

Fixes Alive

1 par

dtcxzyw

Alive2 proo `smt-to`): https://ali

main (

1 parent eb

[InstCombine] fo || (a != c && == (b != c) (#

zjaffal authored on

resolves #92966

alive proof https://alive2.llvm.c

main (#94915)

1 parent 3ae6755 comm

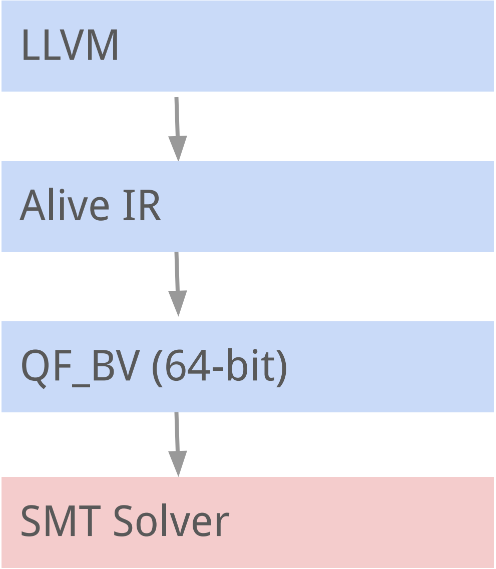
[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e



LLVM

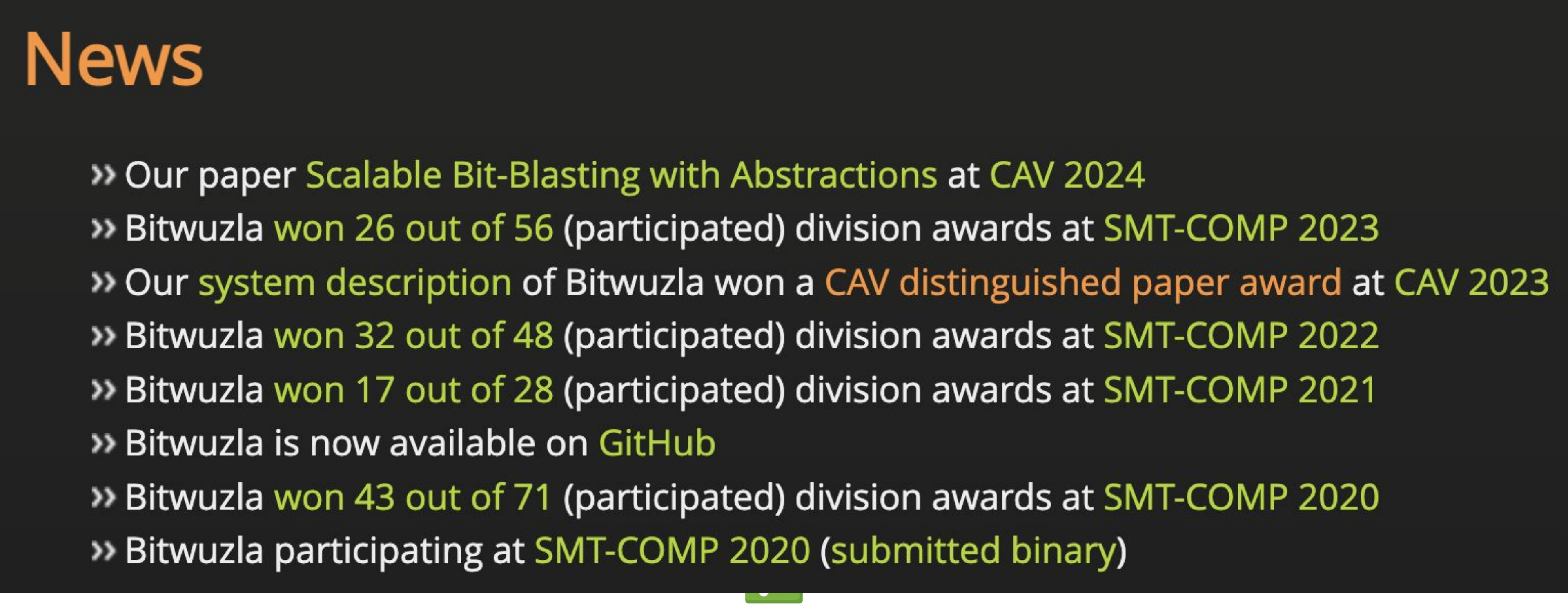
Alive IR

QF_BV (64-bit)

SMT Solver

Bitwuzla

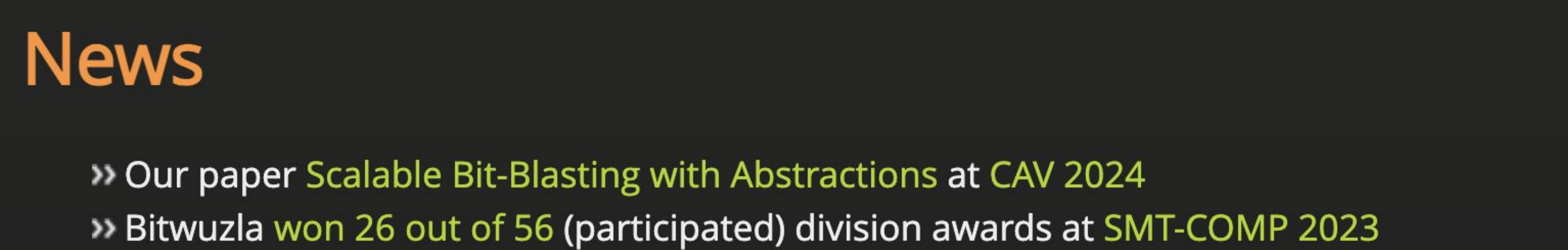
An SMT solver for bit-vectors, floating-points, arrays and uninterpreted functions.



News

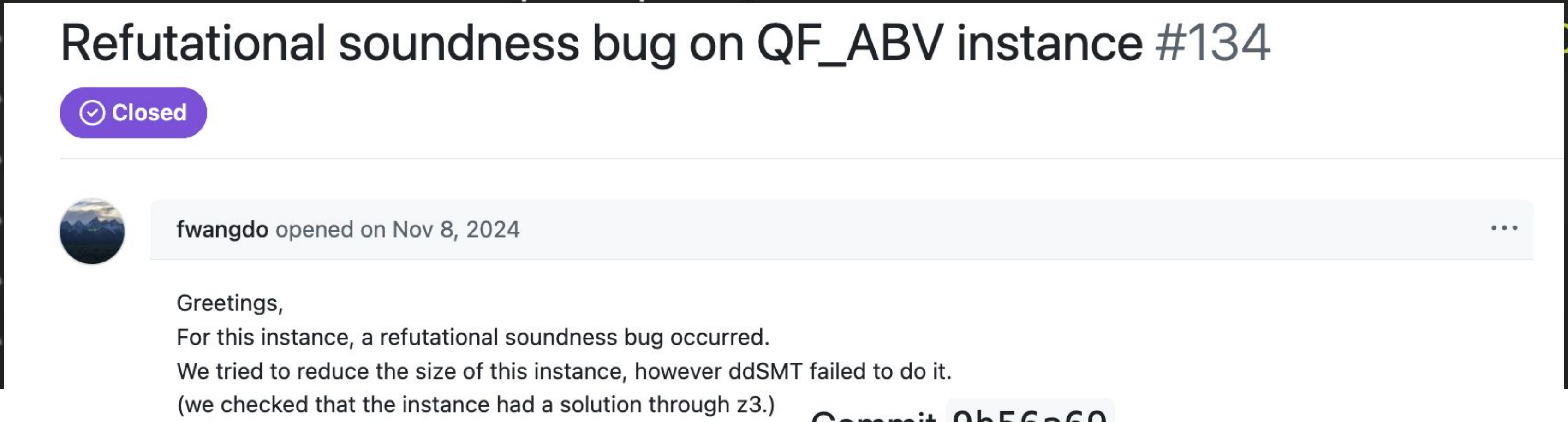
- Our paper Scalable Bit-Blasting with Abstractions at CAV 2024
- Bitwuzla won 26 out of 56 (participated) division awards at SMT-COMP 2023
- Our system description of Bitwuzla won a CAV distinguished paper award at CAV 2023
- Bitwuzla won 32 out of 48 (participated) division awards at SMT-COMP 2022
- Bitwuzla won 17 out of 28 (participated) division awards at SMT-COMP 2021
- Bitwuzla is now available on GitHub
- Bitwuzla won 43 out of 71 (participated) division awards at SMT-COMP 2020
- Bitwuzla participating at SMT-COMP 2020 (submitted binary)

Bitwuzla



News

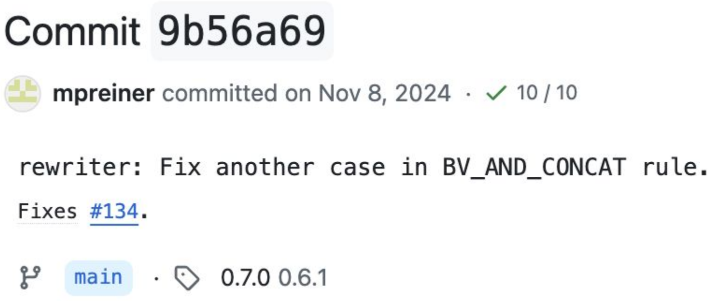
- » Our paper Scalable Bit-Blasting with Abstractions at CAV 2024
- » Bitwuzla won 26 out of 56 (participated) division awards at SMT-COMP 2023
- » 023



Refutational soundness bug on QF_ABV instance #134

fwangdo opened on Nov 8, 2024

Greetings,
For this instance, a refutational soundness bug occurred.
We tried to reduce the size of this instance, however ddSMT failed to do it.
(we checked that the instance had a solution through z3.)



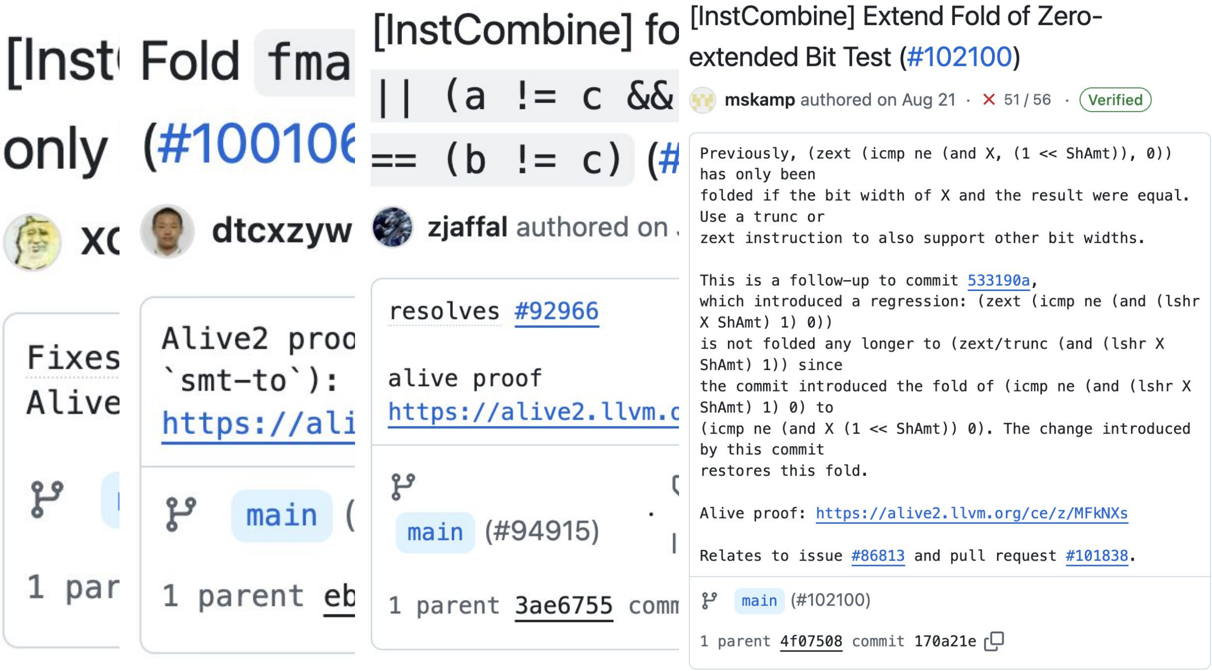
Commit 9b56a69

mpreiner committed on Nov 8, 2024 · ✓ 10 / 10

```
rewriter: Fix another case in BV_AND_CONCAT rule.

Fixes #134.
```

Bitwuzla

Alive is Awesome! How Does It Work?



[Inst Fold fma only (#100106

xc dtcxzyw

Fixes Alive

1 par

Alive2 proo `smt-to`): https://ali

main (

1 parent eb

[InstCombine] fo || (a != c && == (b != c) (#

zjaffal authored on

resolves #92966

alive proof https://alive2.llvm.c

main (#94915)

1 parent 3ae6755 comm

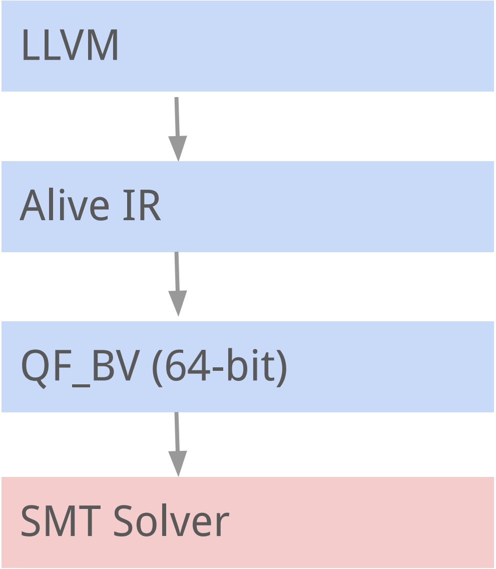
[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

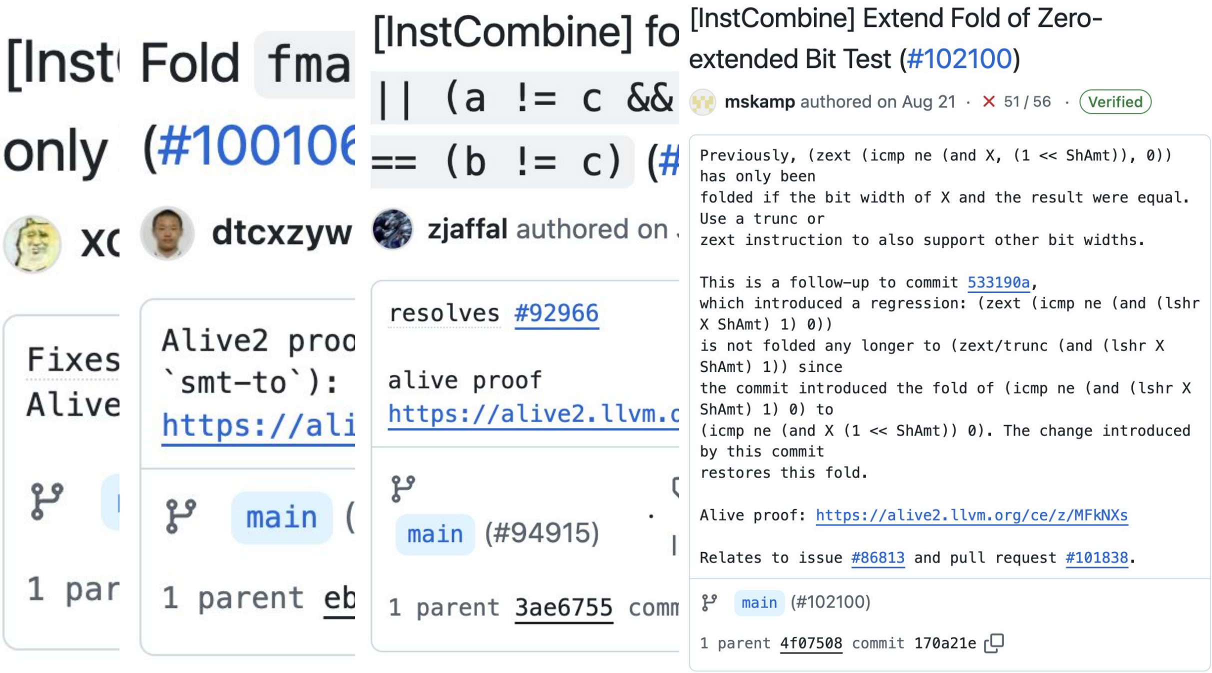
```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e



Alive is Awesome! How Does It Work?



[Inst

only

xc

Fixes
Alive

1 par

Fold fma

(#10010

dtcxzyw

Alive2 proo
`smt-to`):
https://ali

main (

1 parent eb

[InstCombine] fo
|| (a != c &&
== (b != c) (#

zjaffal authored on

resolves #92966

alive proof
https://alive2.llvm.c

main (#94915)

1 parent 3ae6755 comm

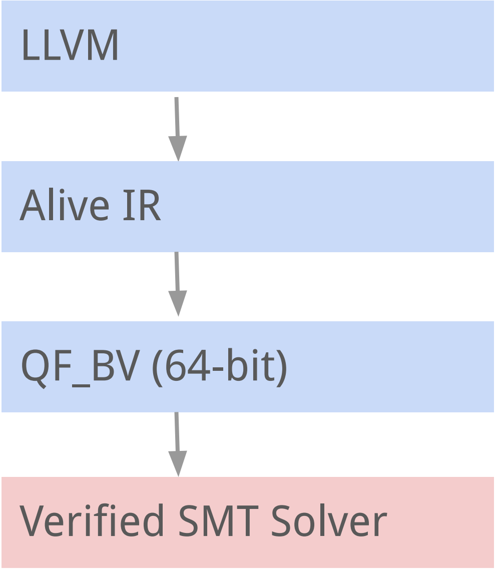
[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e



LLVM

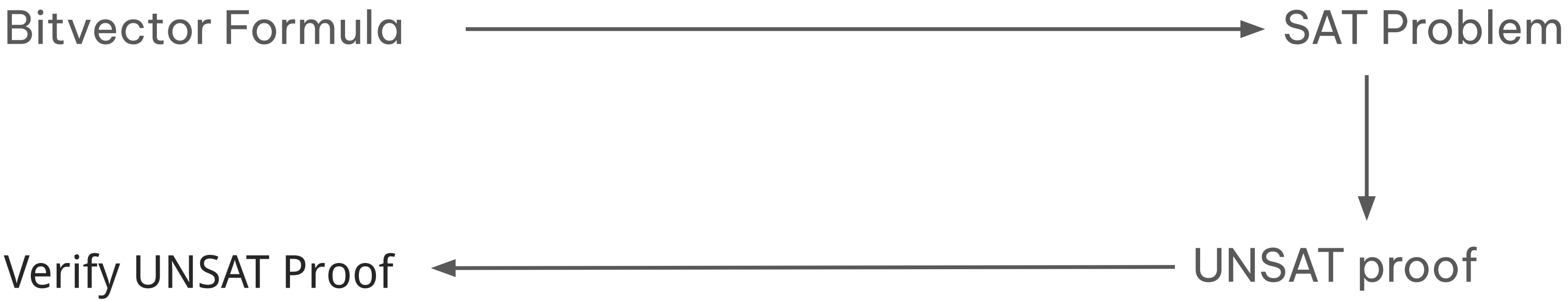
Alive IR

QF_BV (64-bit)

Verified SMT Solver

SMT Solver Proofs Need Arbitrary Width Theorems

Verified Fixed Width Bitvector Solver: `bv_decide`



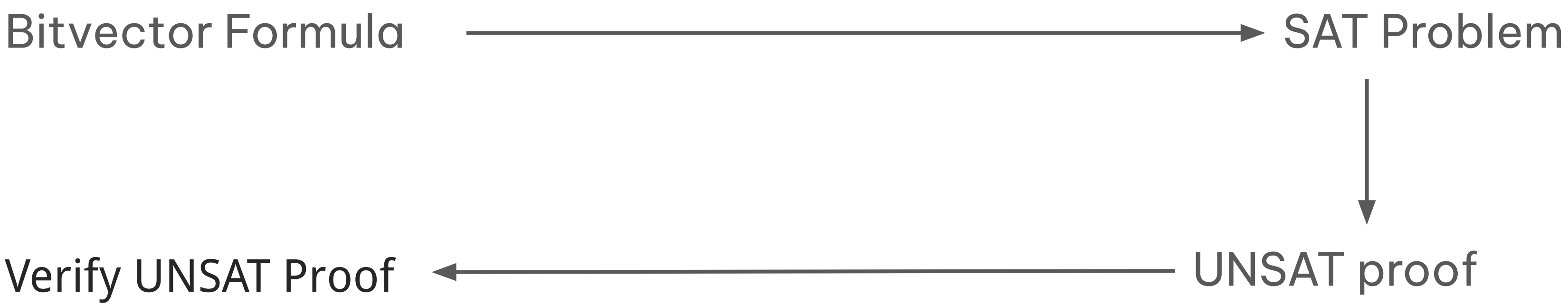
Bitvector Formula

SAT Problem

UNSAT proof

Verify UNSAT Proof

Verified Fixed Width Bitvector Solver: `bv_decide`



```
theorem unsat_of_verifyBVExpr_eq_true (bv : BVLogicalExpr) (c : String)
   (h : verifyBVExpr bv c = true) : ∀ (f : Assignment), eval f bv = false
```

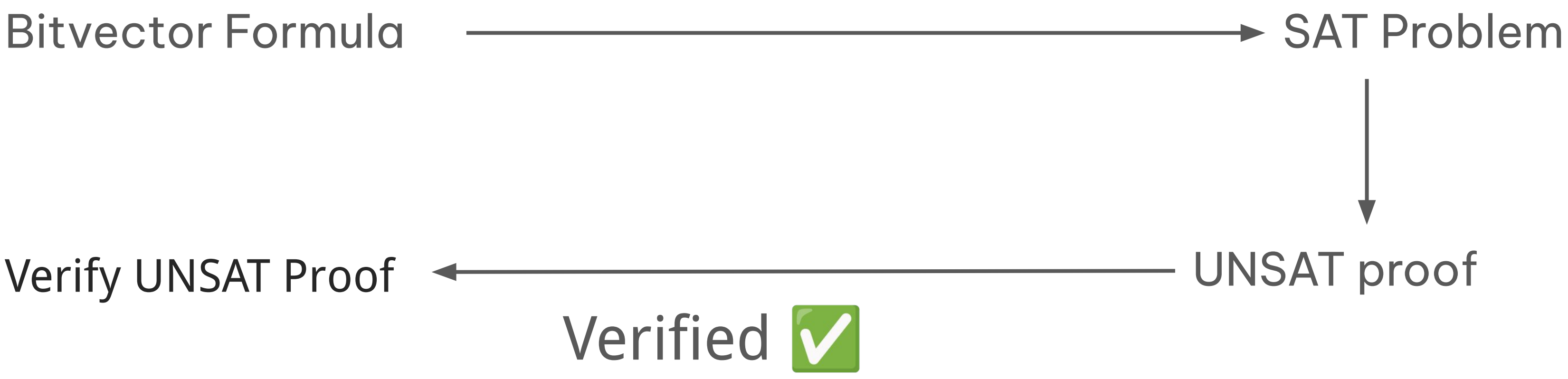
RAT Proof Certificates: Pure Literal Elimination

Resolution Asymmetric Tautology (RAT) [IJCAR 2012]

Given a clause $C = (l_1 \vee \cdots \vee l_k)$ and a CNF formula F:

- $\overline{C}$ denotes the conjunction of its negated literals $(\bar{l}_1) \wedge \cdots \wedge (\bar{l}_k)$
- $F \vdash_1 \epsilon$ denotes that unit propagation on F derives a conflict
- C is an asymmetric tautology w.r.t. F if and only if $F \wedge \overline{C} \vdash_1 \epsilon$
- C is a resolution asymmetric tautology on $l \in C$ w.r.t. F iff for all resolvents $C \diamond D$ with $D \in F$ and $\bar{l} \in D$ holds that $F \wedge \overline{C \diamond D} \vdash_1 \epsilon$

Verified Fixed Width Bitvector Solver: `bv_decide`



Bitvector Formula

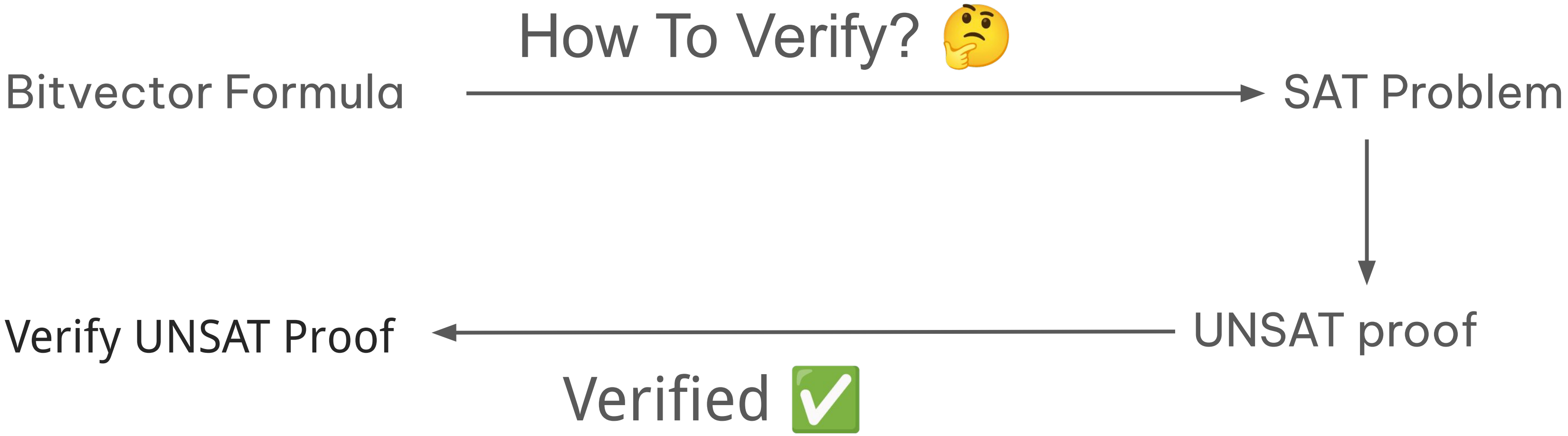
SAT Problem

UNSAT proof

Verify UNSAT Proof

Verified ✅

Verified Fixed Width Bitvector Solver: `bv_decide`



How To Verify? 🤔

Bitvector Formula

SAT Problem

UNSAT proof

Verify UNSAT Proof

Verified ✅

Verified Fixed Width Bitvector Solver: `bv_decide`

How To Verify? 🤔

Bitvector Formula → SAT Problem

```
011b + 100b
= 3 + 4
= 7
= 111b
```

Verified Fixed Width Bitvector Solver: `bv_decide`

Bitvector Formula

How To Verify? 🤔

SAT Problem

```
011b + 100b (bits / booleans ✅)
= 3 + 4
= 7 (natural numbers ❌)
= 111b
```

Verified Fixed Width Bitvector Solver: `bv_decide`

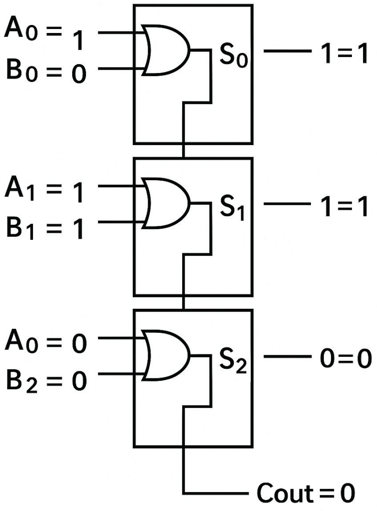
How To Verify? 🤔

Bitvector Formula → SAT Problem

```
011b + 100b (bits / booleans ✅)
= 3 + 4
= 7 (natural numbers ❌)
= 111b
```

→

3-BIT FULL ADDER



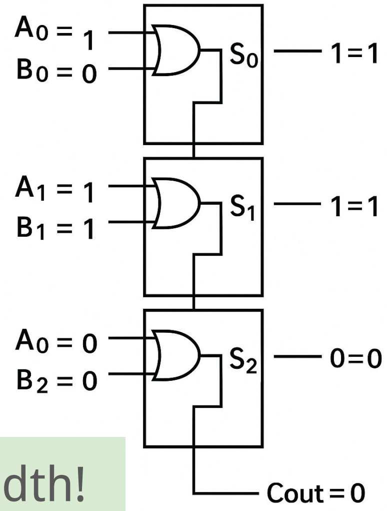
Verified Fixed Width Bitvector Solver: `bv_decide`

How To Verify? 🤔

Bitvector Formula → SAT Problem

`011b + 100b` (bits / booleans ✅)
`= 3 + 4`
`= 7` (natural numbers ❌)
`= 111b`

3-BIT FULL ADDER



In General, We Must Prove This Correct For Arbitrary Width!

Alive is Awesome! How Does It Work?

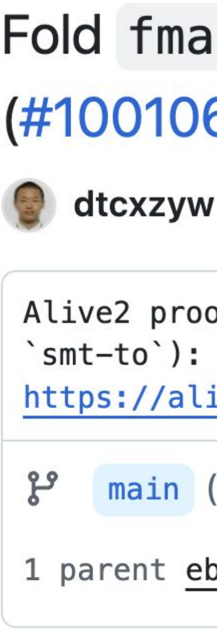
[Inst

only

xc

Fixes
Alive

1 par



Fold fma

(#10010€

dtcxzyw

```
Alive2 proo
`smt-to`):
https://ali
```

main (

1 parent eb

[InstCombine] fo

`|| (a != c &&`

`== (b != c)` (#

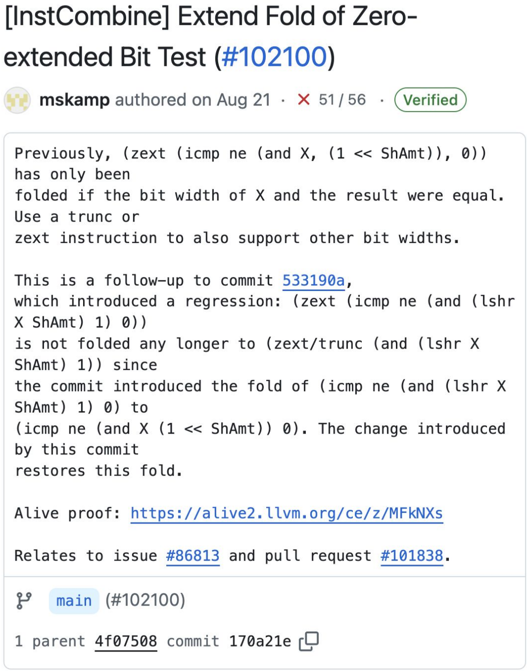
zjaffal authored on

resolves #92966

alive proof
https://alive2.llvm.c

main (#94915)

1 parent 3ae6755 comm



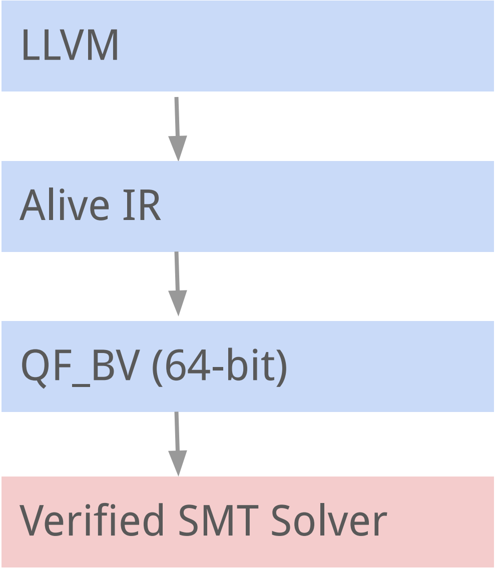
[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e



LLVM

Alive IR

QF_BV (64-bit)

Verified SMT Solver

In General, We Must Prove This Correct For Arbitrary Width!

Gigantic Bitvectors Are Becoming Common!

Arm SVE: 2048 Bit Width Vectors
Fully Homomorphic Encryption: Wide Registers

Gigantic Bitvectors Are Becoming Common!

Arm SVE: 2048 Bit Width Vectors

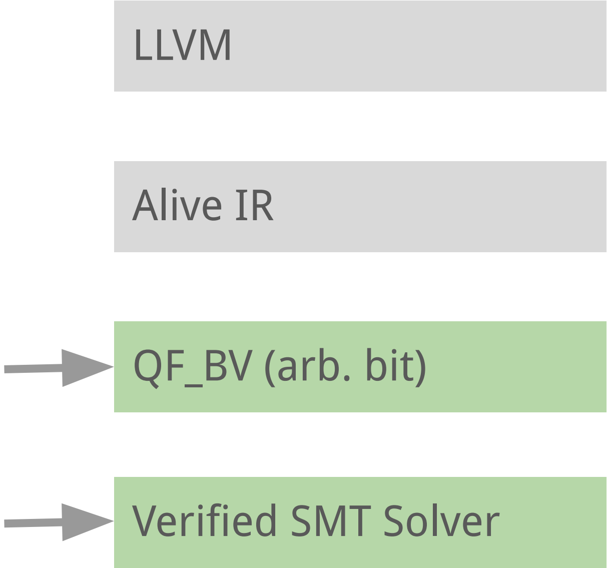
Fully Homomorphic Encryption: Wide Registers

I Sleep Better At Night With an Arbitrary Width Proof

Just Prove The Arbitrary Width Theorems 🔥

Algorithms for Arbitrary Bitwidth Proofs

`bv_automata:` Automata Theory
`bv_mba:` 1bit to Nbit generalization


LLVM

Alive IR

QF_BV (arb. bit)

Verified SMT Solver

Just Prove The Arbitrary Width Theorems 🔥

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

...	x5	x4	x3	x2	x1	**x0**	:	**x**
...	x5	x4	x3	x2	x1	**x0**	:	**x**
							:	**x&x**

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1     x0 :   x
...   x5   x4   x3   x2   x1     x0 :   x
                              x0&x0 : x&x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2   x1     x0 :     x
...   x5   x4   x3   x2   x1     x0 :     x
                              x0&x0 :   x&x
                                  1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
... x5 x4 x3 x2 x1 x0 :     x
... x5 x4 x3 x2 x1 x0 :     x
                x0&x0 :   x&x
                    1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
...   x5   x4   x3   x2      x1    x0 :     x
...   x5   x4   x3   x2      x1    x0 :     x
                          x1&x1 x0&x0 :   x&x
                                    1 : x&x=x
```

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

...	x5	x4	x3	x2	**x1**	x0	:	**x**
...	x5	x4	x3	x2	**x1**	x0	:	**x**
					x1&x1	x0&x0	:	**x&x**
					1	1	:	**x&x=x**

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

...	x5	x4	x3	**x2**	x1	x0	:	**x**
...	x5	x4	x3	**x2**	x1	x0	:	**x**
				x2&x2	x1&x1	x0&x0	:	**x&x**
				1	1	1	:	**x&x=x**

bv_automata: Width-Quantified Problems

```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
         ...    x5    x4    x3    x2    x1    x0  :     x

         ...    x5    x4    x3    x2    x1    x0  :     x

           ...................x2&x2 x1&x1 x0&x0  :   x&x

    .....1.....1.....1.....1.....1     1     1  : x&x=x
```

bv_automata: Width-Quantified Problems

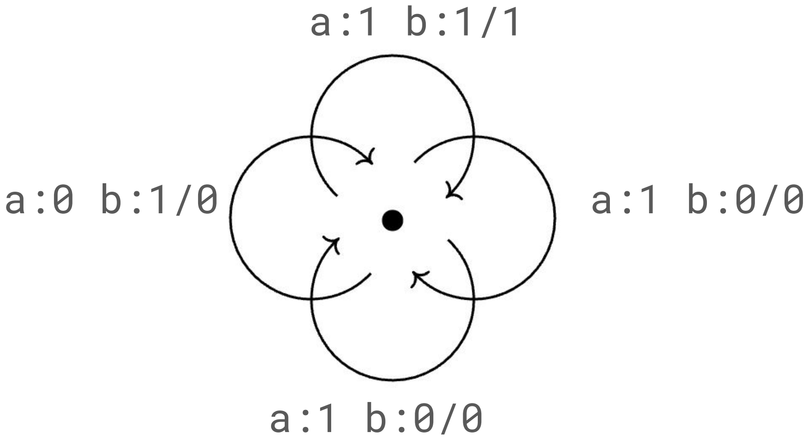
```
theorem and_idem: ∀ (w : Nat) (x : BitVec w), x & x = x := by bv_automata
```

```
       ...    x5    x4    x3    x2    x1    x0 :      x

       ...    x5    x4    x3    x2    x1    x0 :      x

         ...................x2&x2 x1&x1 x0&x0 :    x&x

    .....1.....1.....1.....1.....1     1     1 : x&x=x
```

Does **x&x=x** produce an infinite sequence of 1s?

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



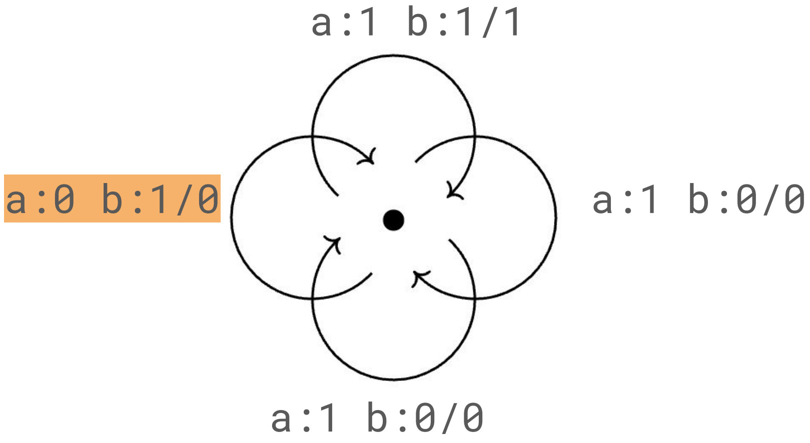
: **a**

: **b**

: **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



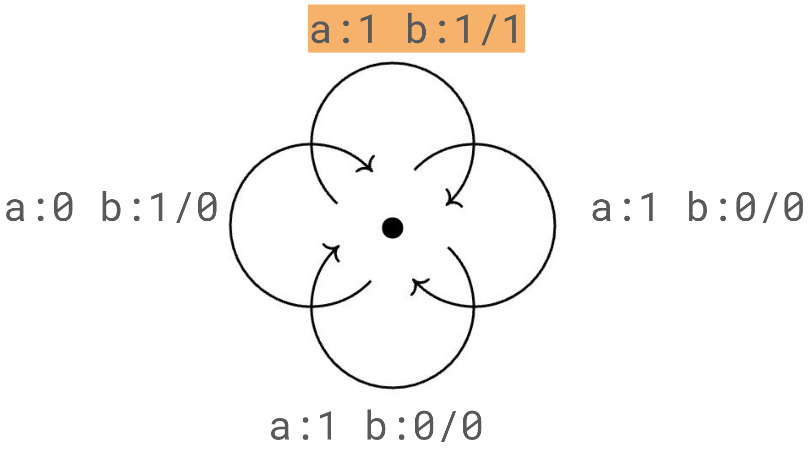
0 : **a**

1 : **b**

0 : **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



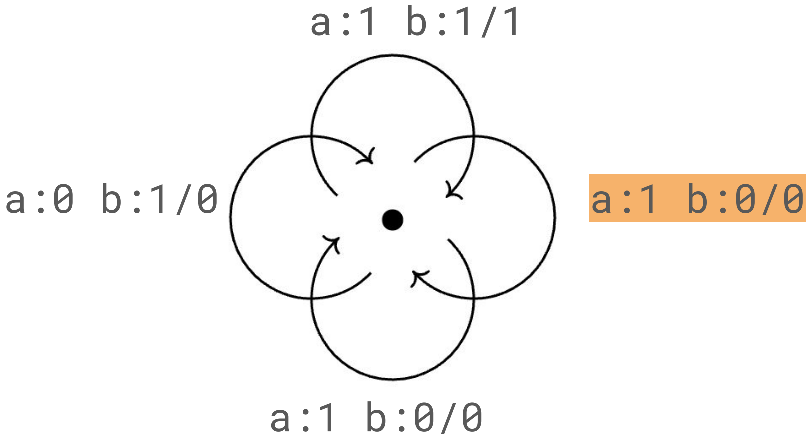
1 0 : **a**

1 1 : **b**

1 0 : **a&b**

Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**



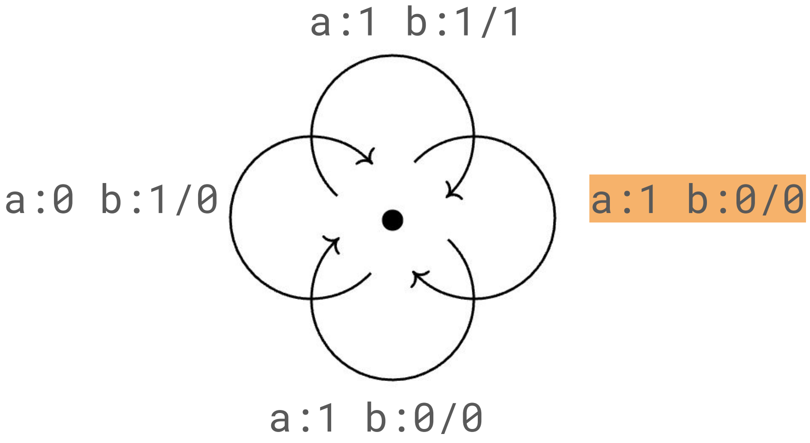
1 1 0 : **a**

0 1 1 : **b**

0 1 0 : **a&b**

Does x&x=x produce infinite sequence of 1s?

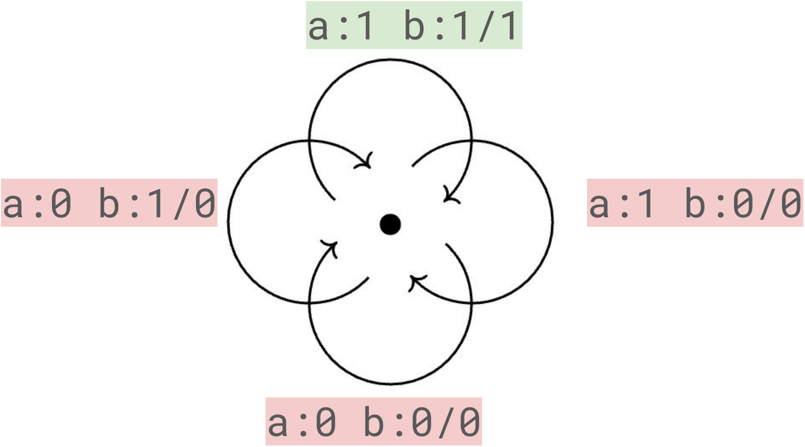
Automata for **a&b:**



1	1	0	:	**a**
0	1	1	:	**b**
0	1	0	:	**a&b**

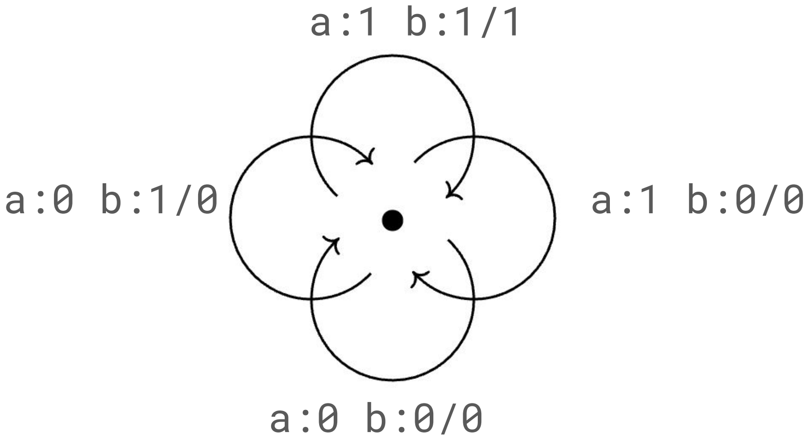
Does x&x=x produce infinite sequence of 1s?

Automata for **a&b:**

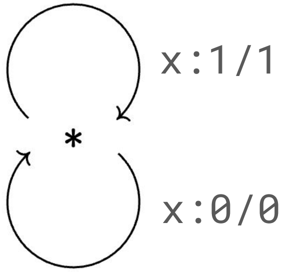


Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a&b:**

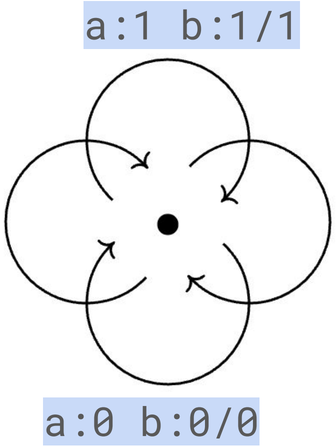


Automata for **x:**

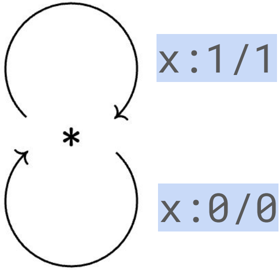


Does x&x=x Produce An Infinite Sequence of 1s?

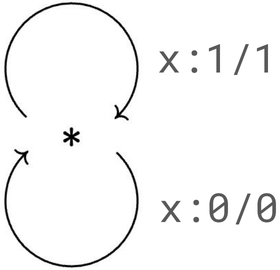
Automata for **a&b:**



Automata for **x:**

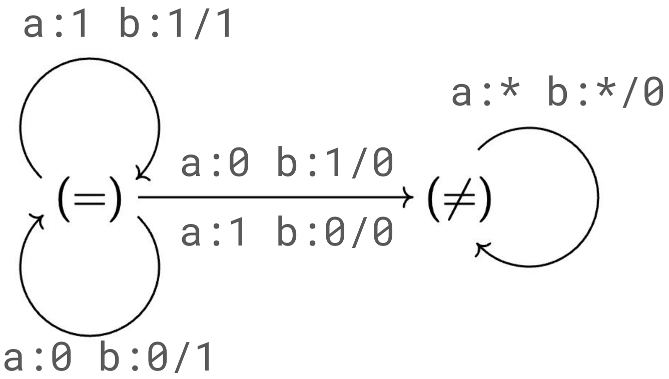


Automata for **x&x:**



Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a=b:**



a:1 b:1/1

a:* b:*/0

a:0 b:1/0

a:1 b:0/0

a:0 b:0/1

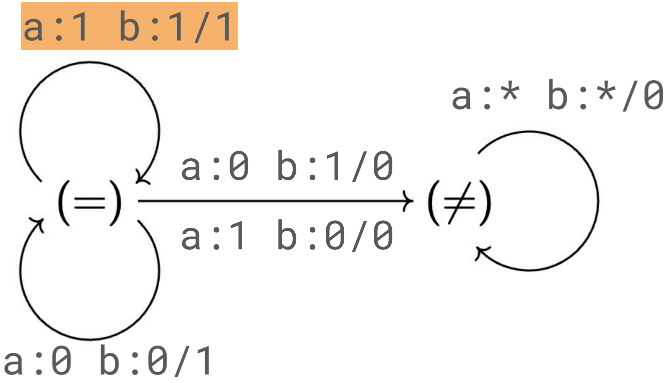
: **a**

: **b**

: **a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

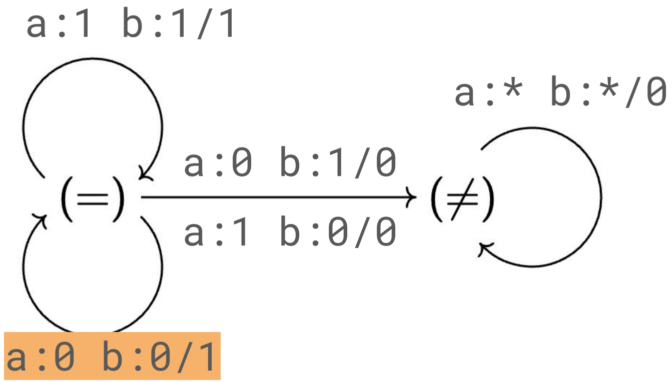
Automata for **a=b:**



1	:	**a**
1	:	**b**
1	:	**a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

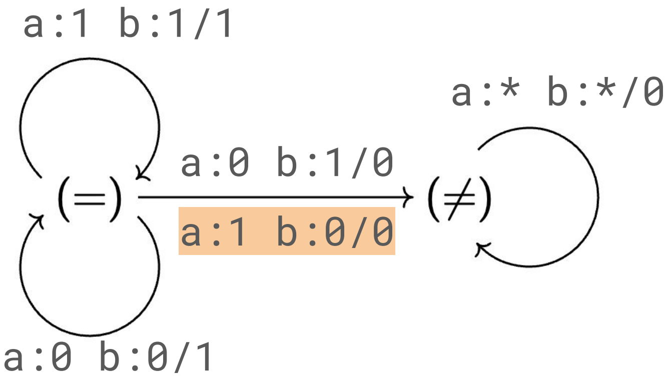
Automata for **a=b:**



```
0 1 :   a
0 1 :   b
1 1 : a=b
```

Does x&x=x Produce An Infinite Sequence of 1s?

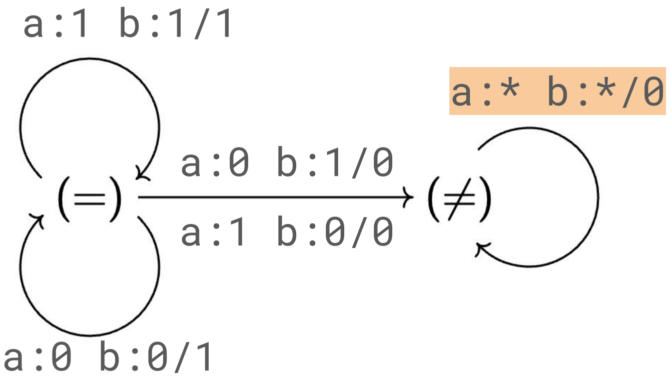
Automata for **a=b:**



```
0 0 1 :   a
1 0 1 :   b
0 1 1 : a=b
```

Does x&x=x Produce An Infinite Sequence of 1s?

Automata for **a=b:**



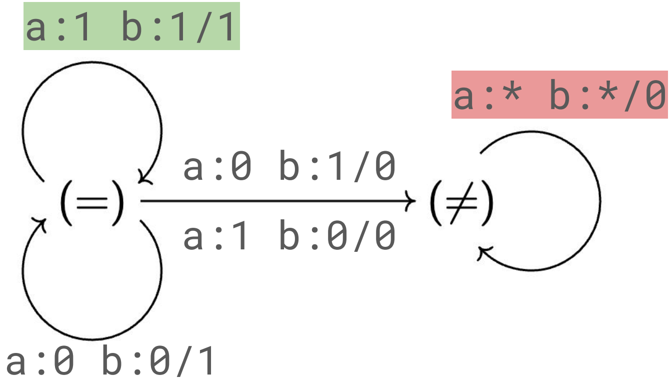
1 0 0 1 : **a**

1 1 0 1 : **b**

0 0 1 1 : **a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

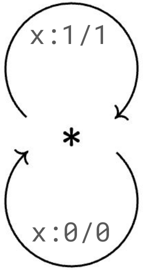
Automata for **a=b:**



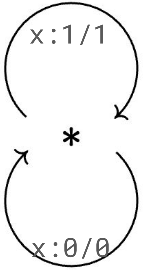
1	0	0	1	:	**a**
1	1	0	1	:	**b**
0	0	1	1	:	**a=b**

Does x&x=x Produce An Infinite Sequence of 1s?

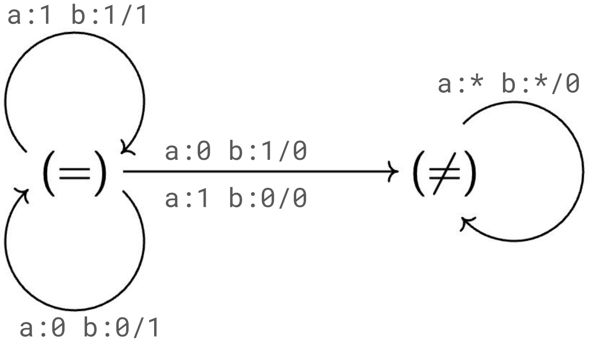
Automata for **x&x**:



Automata for **x**:

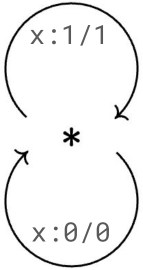


Automata for **a=b**:

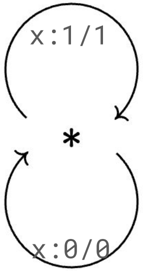


Does x&x=x Produce An Infinite Sequence of 1s?

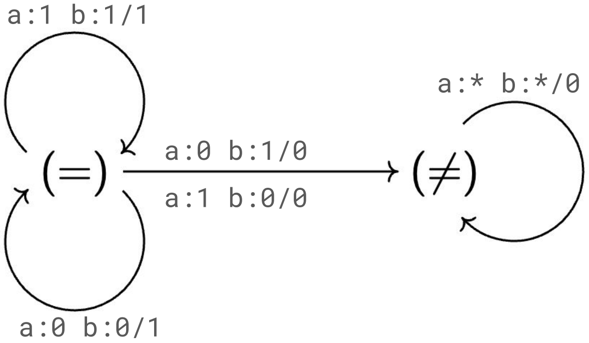
Automata for **x&x:**



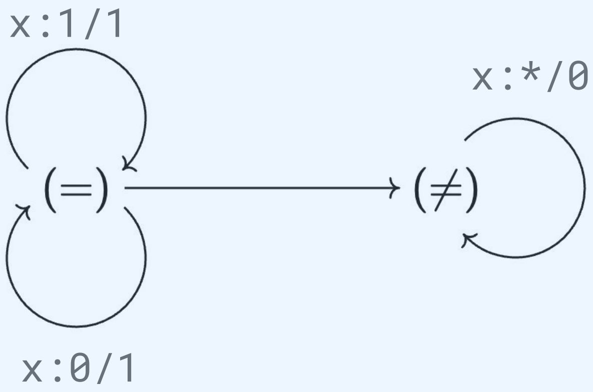
Automata for **x:**



Automata for **a=b:**

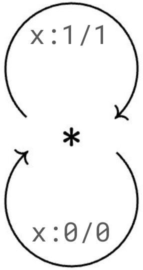


Automata for **x&x=x:**

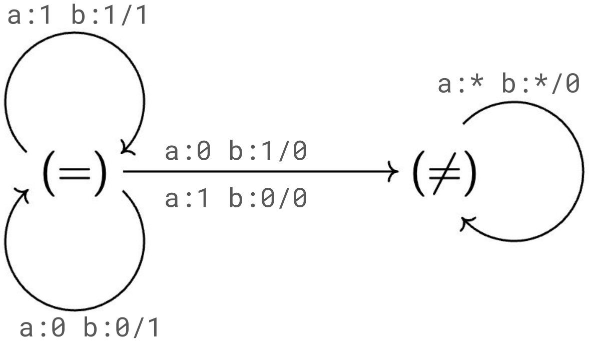


Does x&x=x Produce An Infinite Sequence of 1s?

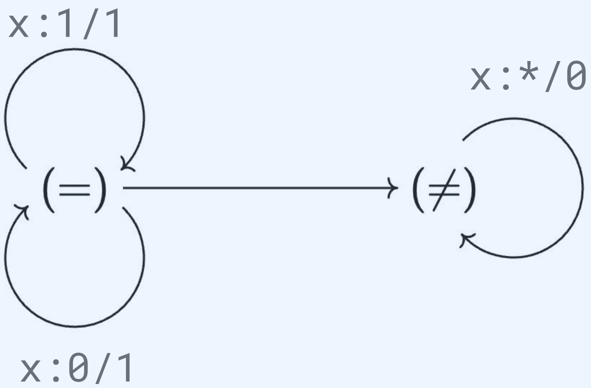
Automata for **x&x:**



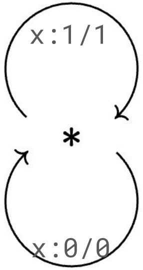
Automata for **a=b:**



Automata for **x&x=x:**



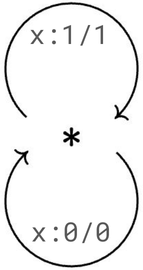
Automata for **x:**



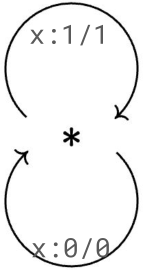
Does Automata for **x&x=x** Always Produce 1s?

Does x&x=x Produce An Infinite Sequence of 1s?

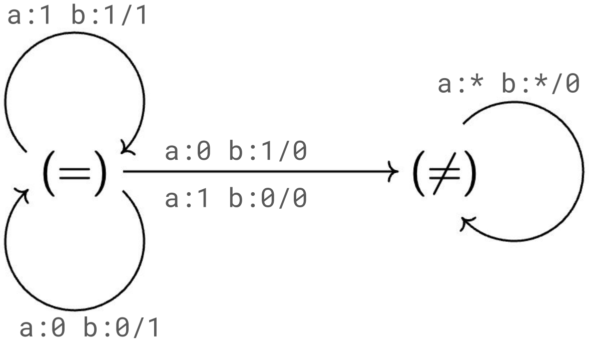
Automata for **x&x:**



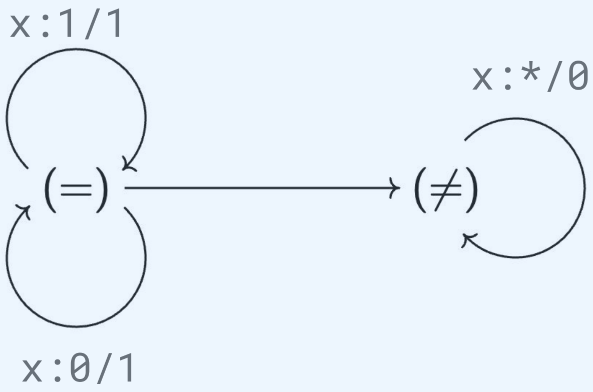
Automata for **x:**



Automata for **a=b:**



Automata for **x&x=x:**



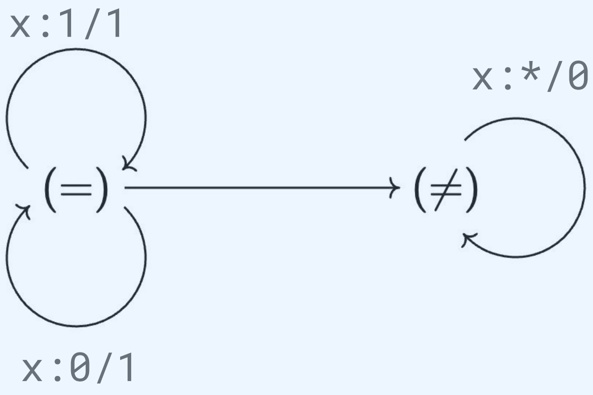
YES: Automata for **x&x=x** Always Produce 1s!

Does x&x=x Produce An Infinite Sequence of 1s?

```
theorem and_idem: ∀ (w : Nat)(x : BitVec w),
    x & x = x := by bv_automata
```

```
      ...x1     x0 :     x

      ...x1     x0 :     x

   ...x1&x1 x0&x0 :   x&x

       ...1      1 : x&x=x
```

Automata for **x&x=x:**



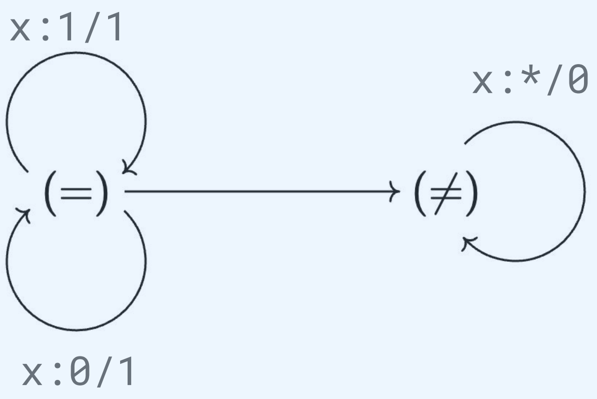
YES: Automata for **x&x=x** Always Produce 1s!

Does x&x=x Produce An Infinite Sequence of 1s?

```
theorem and_idem: ∀ (w : Nat)(x : BitVec w),
    x & x = x := by bv_automata
```

```
      ...x1     x0 :     x
      ...x1     x0 :     x
   ...x1&x1  x0&x0 :   x&x
       ...1      1 : x&x=x
```

Automata for **x&x=x:**



YES: Automata for **x&x=x** Always Produce 1s!

Does x&x=x Produce An Infinite Sequence of 1s?

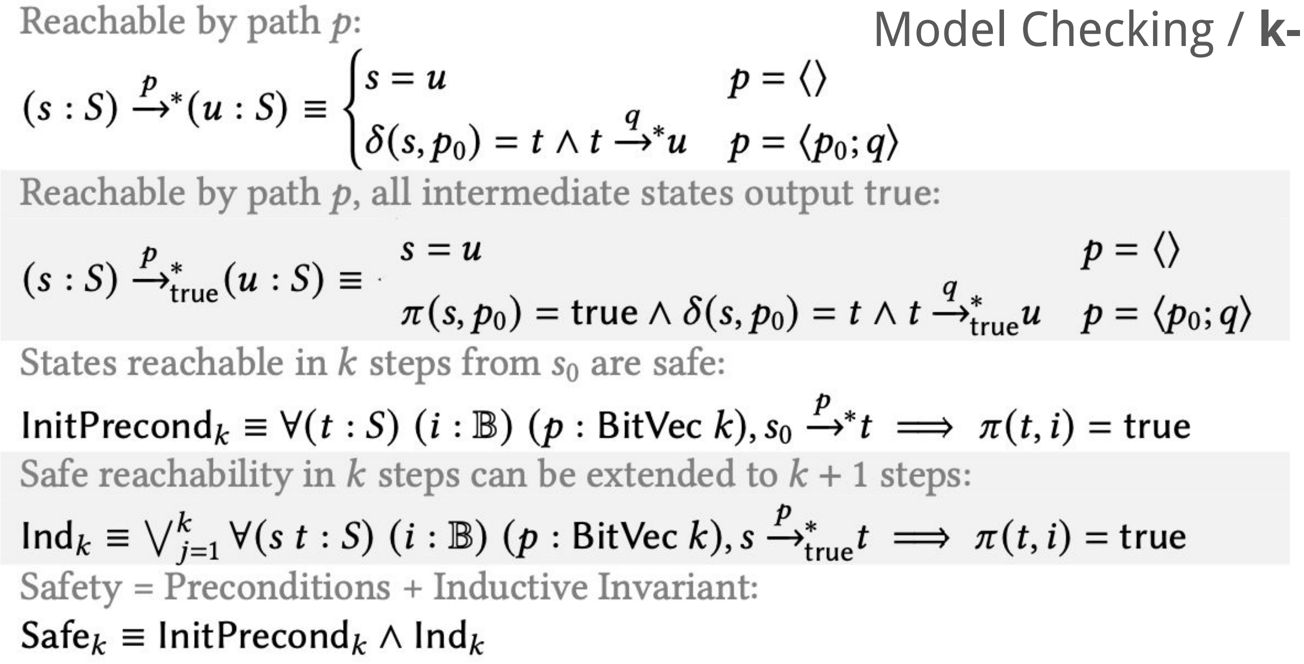
YES: Automata for **x&x=x** Always Produce 1s! 🤔

Does x&x=x Produce An Infinite Sequence of 1s?

YES: Automata for **x&x=x** Always Produce 1s! 🤔

Model Checking / **k-induction**

🥰



Reachable by path p:

$$(s : S) \xrightarrow{p}{}^{*} (u : S) \equiv \begin{cases} s = u & p = \langle\rangle \\ \delta(s, p_0) = t \land t \xrightarrow{q}{}^{*} u & p = \langle p_0; q\rangle \end{cases}$$

Reachable by path p, all intermediate states output true:

$$(s : S) \xrightarrow{p}{}^{*}_{\mathsf{true}} (u : S) \equiv \begin{cases} s = u & p = \langle\rangle \\ \pi(s, p_0) = \mathsf{true} \land \delta(s, p_0) = t \land t \xrightarrow{q}{}^{*}_{\mathsf{true}} u & p = \langle p_0; q\rangle \end{cases}$$

States reachable in k steps from s_0 are safe:

$$\mathsf{InitPrecond}_k \equiv \forall (t : S)\ (i : \mathbb{B})\ (p : \mathsf{BitVec}\ k), s_0 \xrightarrow{p}{}^{*} t \implies \pi(t, i) = \mathsf{true}$$

Safe reachability in k steps can be extended to $k + 1$ steps:

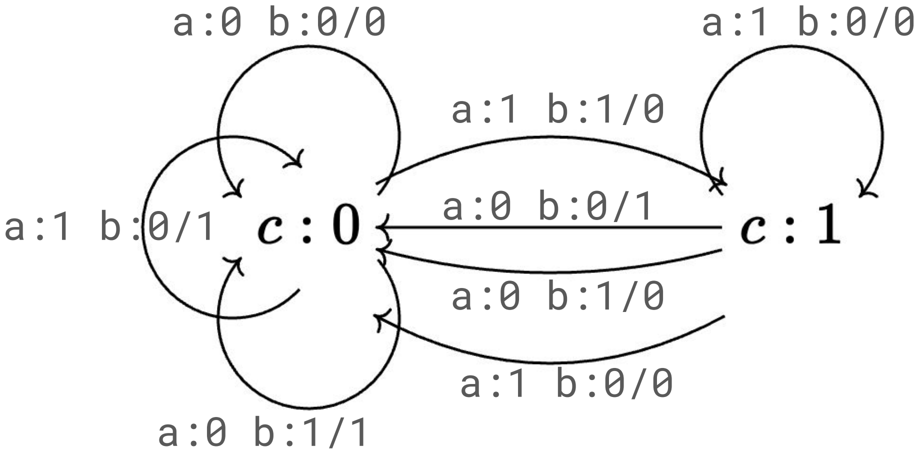
$$\mathsf{Ind}_k \equiv \bigvee_{j=1}^{k} \forall (s\ t : S)\ (i : \mathbb{B})\ (p : \mathsf{BitVec}\ k), s \xrightarrow{p}{}^{*}_{\mathsf{true}} t \implies \pi(t, i) = \mathsf{true}$$

Safety = Preconditions + Inductive Invariant:

$$\mathsf{Safe}_k \equiv \mathsf{InitPrecond}_k \land \mathsf{Ind}_k$$

Automata for a+b :

Automata for **a+b:**



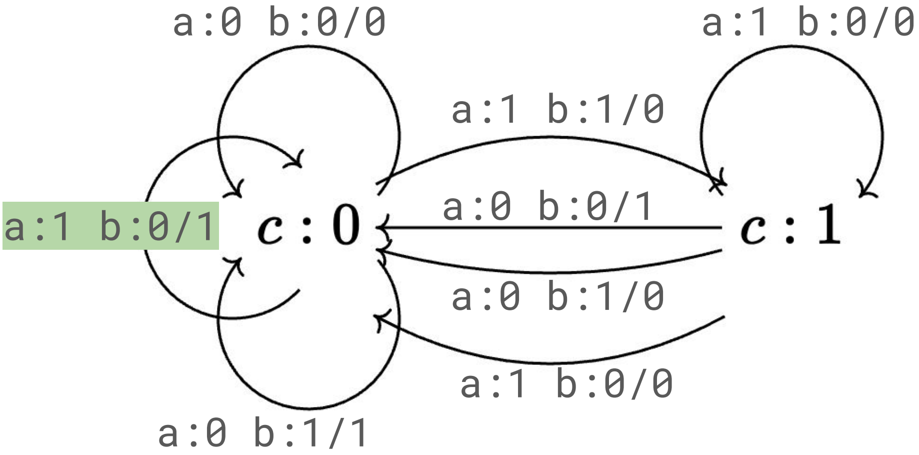
: **a**

: **b**

: **a+b**

Automata for a+b :

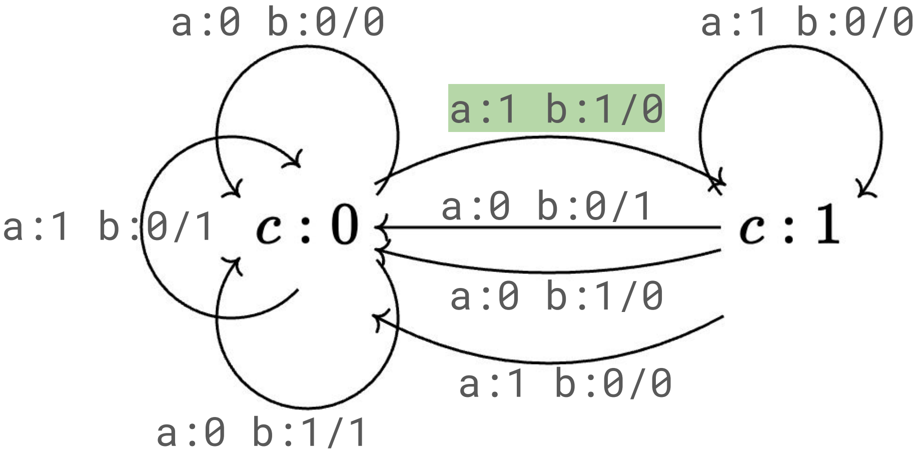
Automata for **a+b:**



1:	**a**
0:	**b**
1:	**a+b**

Automata for a+b :

Automata for **a+b:**



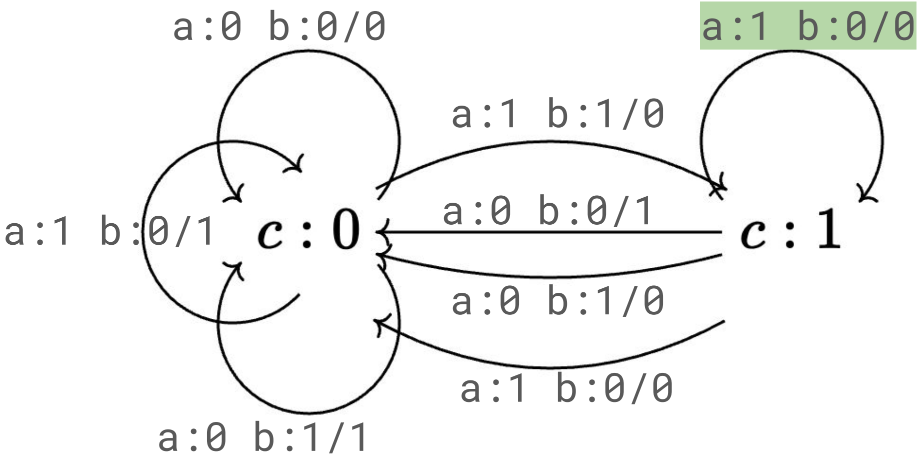
1 1: **a**

1 0: **b**

0 1: **a+b**

Automata for a+b :

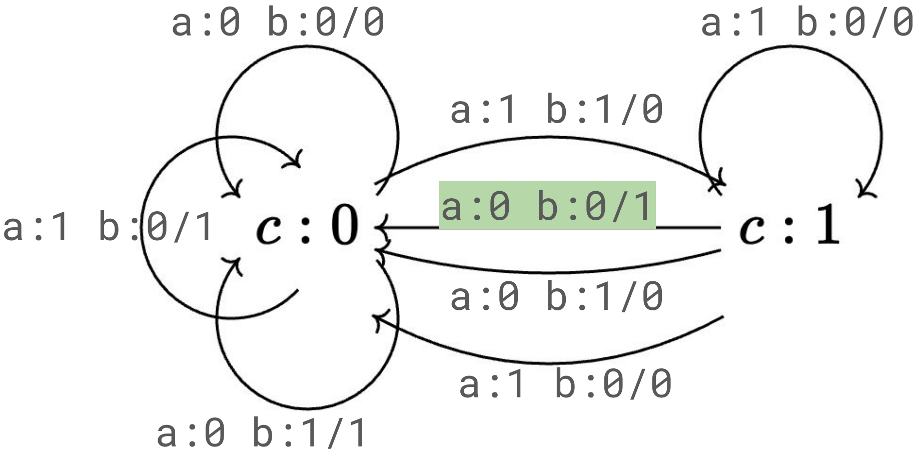
Automata for **a+b:**



```
1
1 1 1:    a
0 1 0:    b
0 0 1:  a+b
```

Automata for a+b :

Automata for **a+b:**



```
1 1
0 1 1 1:   a
0 0 1 0:   b
1 0 0 1: a+b
```

Automata for P ∧ Q:

Recall Automata for equality: Told us if equality was **true up to the index**

```
1 0 0 1 :   a
1 1 0 1 :   b
0 0 1 1 : a=b
```

Automata for **P ∧ Q**:

```
 1:    P
 1:    Q
1:  P∧Q
```

Automata for P ∧ Q:

Recall Automata for equality: Told us if equality was **true up to the index**

```
1 0 0 1 :   a
1 1 0 1 :   b
0 0 1 1 : a=b
```

Automata for **P ∧ Q**:

```
1 1:    P
1 1:    Q
1 1:  P∧Q
```

Automata for P ∧ Q:

Recall Automata for equality: Told us if equality was **true up to the index**

```
1 0 0 1 :   a
1 1 0 1 :   b
0 0 1 1 : a=b
```

Automata for **P ∧ Q**:

```
 1 1 1:   P
 0 1 1:   Q
0 1 1: P∧Q
```

Automata for P ∧ Q:

Recall Automata for equality: Told us if equality was **true up to the index**

```
1 0 0 1 :   a
1 1 0 1 :   b
0 0 1 1 : a=b
```

Automata for **P ∧ Q :**

```
1 1 1 1:   P
1 0 1 1:   Q
0 0 1 1: P∧Q
```

Automata for P ∧ Q:

Recall Automata for equality: Told us if equality was **true up to the index**

```
1 0 0 1 :   a
1 1 0 1 :   b
0 0 1 1 : a=b
```

Automata for **P ∧ Q:**

```
0 1 1 1 1:   P
0 1 0 1 1:   Q
0 0 0 1 1: P∧Q
```

Automata for P ∧ Q:

Recall Automata for equality: Told us if equality was **true up to the index**

```
1 0 0 1 :   a
1 1 0 1 :   b
0 0 1 1 : a=b
```

Automata for **P ∧ Q :**

```
0 1 1 1 1:    P
0 1 0 1 1:    Q
0 0 0 1 1:  P∧Q
```

Automata for **P∨Q :**

```
0 1 1 1 1:    P
0 1 0 1 1:    Q
0 0 0 1 1:  P∨Q
```

What Is Automata Representable?

- Bitwise Operations, Equality

- Addition (Build Add-Carry Circuit)

- Negation (`-x = !x + 1`)

- Multiplication by Constants: `3 * x = x + x + x`

- Boolean Combinations of Conditions: and, or, not.

- Left Shift: `a <<< 2 = a * 4 = a + a + a + a`

- Right Shift: `a >>> 1 = b` **if and only if** the bits `b[i]` equals `a[i+1]`:

```
    ∀ aShift, aShift & (..1110) = a → aShift = b
```

What Is Automata Representable? (WIP Extensions)

- Sign Extend, Zero Extend
- Multiple Widths, Append
- IsPowerOf2?

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

bv_mba： Mixed-Boolean-Arithmetic

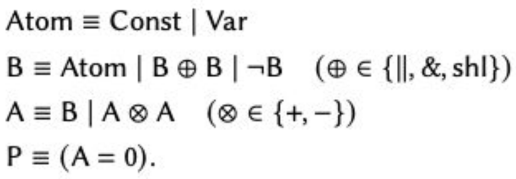
```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

bv_mba: Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

bv_mba : Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```



$$\mathrm{Atom} \equiv \mathrm{Const} \mid \mathrm{Var}$$

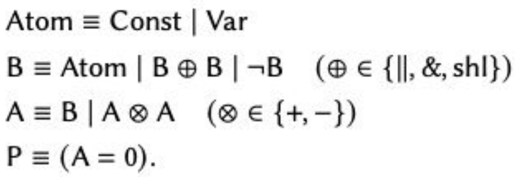
$$\mathrm{B} \equiv \mathrm{Atom} \mid \mathrm{B} \oplus \mathrm{B} \mid \neg \mathrm{B} \quad (\oplus \in \{\|, \&, \mathrm{shl}\})$$

$$\mathrm{A} \equiv \mathrm{B} \mid \mathrm{A} \otimes \mathrm{A} \quad (\otimes \in \{+, -\})$$

$$\mathrm{P} \equiv (\mathrm{A} = 0).$$

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x.toInt + y.toInt - (x ^^^ y).toInt - 2 * (x &&& y).toInt = 0
```

bv_mba： Mixed-Boolean-Arithmetic



$$\mathrm{Atom} \equiv \mathrm{Const} \mid \mathrm{Var}$$

$$\mathrm{B} \equiv \mathrm{Atom} \mid \mathrm{B} \oplus \mathrm{B} \mid \neg \mathrm{B} \quad (\oplus \in \{\|, \&, \mathrm{shl}\})$$

$$\mathrm{A} \equiv \mathrm{B} \mid \mathrm{A} \otimes \mathrm{A} \quad (\otimes \in \{+, -\})$$

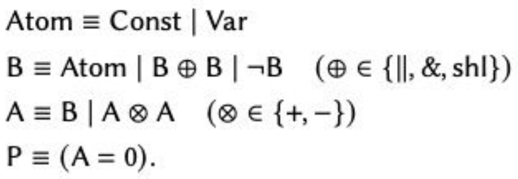
$$\mathrm{P} \equiv (\mathrm{A} = 0).$$

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
```

bv_mba： Mixed-Boolean-Arithmetic



$$\text{Atom} \equiv \text{Const} \mid \text{Var}$$

$$\text{B} \equiv \text{Atom} \mid \text{B} \oplus \text{B} \mid \neg\text{B} \quad (\oplus \in \{\|, \&, \text{shl}\})$$

$$\text{A} \equiv \text{B} \mid \text{A} \otimes \text{A} \quad (\otimes \in \{+, -\})$$

$$\text{P} \equiv (\text{A} = 0).$$

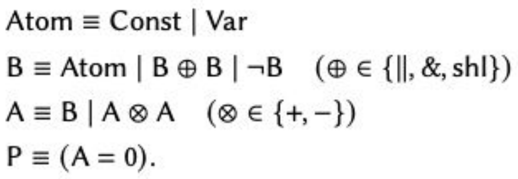
```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
```

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```



$\mathrm{Atom} \equiv \mathrm{Const} \mid \mathrm{Var}$

$B \equiv \mathrm{Atom} \mid B \oplus B \mid \neg B \quad (\oplus \in \{\|, \&, \mathrm{shl}\})$

$A \equiv B \mid A \otimes A \quad (\otimes \in \{+, -\})$

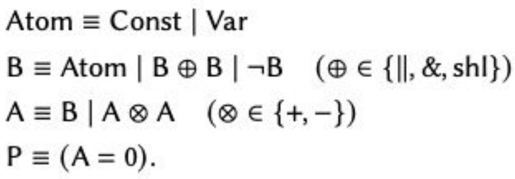
$P \equiv (A = 0).$

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0)
```

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

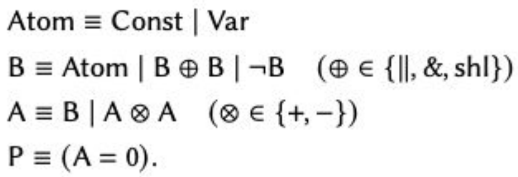


$$\begin{aligned}
&\mathrm{Atom} \equiv \mathrm{Const} \mid \mathrm{Var}\\
&\mathrm{B} \equiv \mathrm{Atom} \mid \mathrm{B} \oplus \mathrm{B} \mid \neg \mathrm{B} \quad (\oplus \in \{\|, \&, \mathrm{shl}\})\\
&\mathrm{A} \equiv \mathrm{B} \mid \mathrm{A} \otimes \mathrm{A} \quad (\otimes \in \{+, -\})\\
&\mathrm{P} \equiv (\mathrm{A} = 0).
\end{aligned}$$

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0)
```

bv_mba： Mixed-Boolean-Arithmetic



$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$

$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg \mathsf{B} \quad (\oplus \in \{||, \&, \mathsf{shl}\})$

$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$

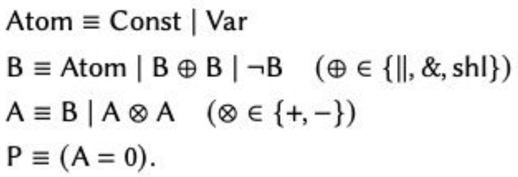
$\mathsf{P} \equiv (\mathsf{A} = 0).$

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0))
```

bv_mba： Mixed-Boolean-Arithmetic



$\mathrm{Atom} \equiv \mathrm{Const} \mid \mathrm{Var}$

$B \equiv \mathrm{Atom} \mid B \oplus B \mid \neg B \quad (\oplus \in \{\|, \&, \mathrm{shl}\})$

$A \equiv B \mid A \otimes A \quad (\otimes \in \{+, -\})$

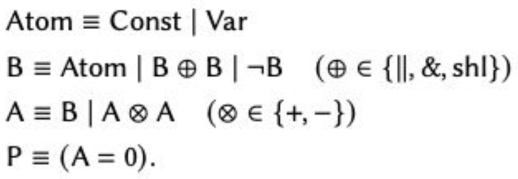
$P \equiv (A = 0).$

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
```

bv_mba： Mixed-Boolean-Arithmetic



$$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$$

$$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg\mathsf{B} \quad (\oplus \in \{\|, \&, \mathsf{shl}\})$$

$$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$$

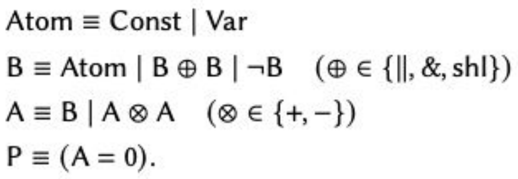
$$\mathsf{P} \equiv (\mathsf{A} = 0).$$

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
```

bv_mba： Mixed-Boolean-Arithmetic



$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$

$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg \mathsf{B} \quad (\oplus \in \{\|, \&, \mathsf{shl}\})$

$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$

$\mathsf{P} \equiv (\mathsf{A} = 0).$

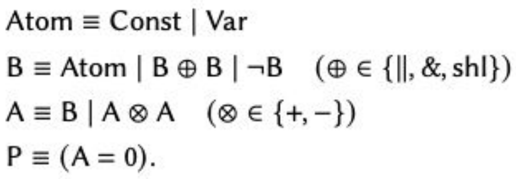
```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1
```

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```



$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$

$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg \mathsf{B} \quad (\oplus \in \{\|, \&, \mathsf{shl}\})$

$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$

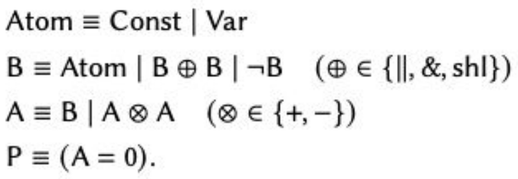
$\mathsf{P} \equiv (\mathsf{A} = 0).$

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1
```

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```



Atom ≡ Const | Var

B ≡ Atom | B ⊕ B | ¬B   (⊕ ∈ {||, &, shl})

A ≡ B | A ⊗ A   (⊗ ∈ {+, −})

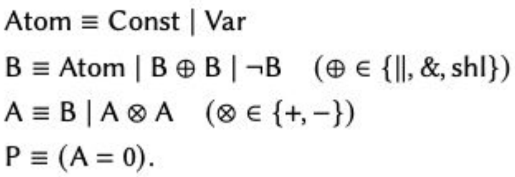
P ≡ (A = 0).

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1 - (x1^y1)
```

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```



$$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$$

$$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg\mathsf{B} \quad (\oplus \in \{\|, \&, \mathsf{shl}\})$$

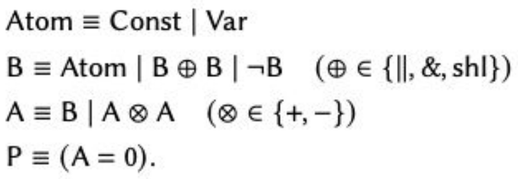
$$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$$

$$\mathsf{P} \equiv (\mathsf{A} = 0).$$

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1 - (x1^y1) -2 * (x1&y1)
```

bv_mba : Mixed-Boolean-Arithmetic



$\text{Atom} \equiv \text{Const} \mid \text{Var}$

$B \equiv \text{Atom} \mid B \oplus B \mid \neg B \quad (\oplus \in \{\|, \&, \text{shl}\})$

$A \equiv B \mid A \otimes A \quad (\otimes \in \{+, -\})$

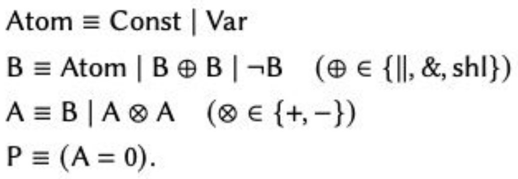
$P \equiv (A = 0).$

```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) : ✅
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1 - (x1^y1) -2 * (x1&y1) = 0
```

bv_mba： Mixed-Boolean-Arithmetic



$$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$$

$$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg\mathsf{B} \quad (\oplus \in \{||, \&, \mathsf{shl}\})$$

$$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$$

$$\mathsf{P} \equiv (\mathsf{A} = 0).$$

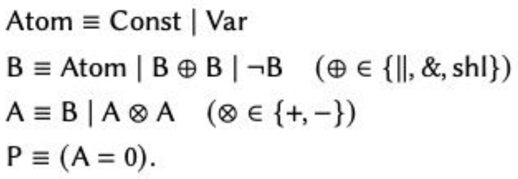
```
theorem add_eq_xor_and (x y : BitVec w) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1 - (x1^y1) -2 * (x1&y1) = 0
2(x0 + y0 - (x0^y0) -2 * (x0&y0) = 0
```

bv_mba： Mixed-Boolean-Arithmetic

```
theorem add_eq_xor_and (x y : BitVec w) : ✅
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```



$\mathsf{Atom} \equiv \mathsf{Const} \mid \mathsf{Var}$

$\mathsf{B} \equiv \mathsf{Atom} \mid \mathsf{B} \oplus \mathsf{B} \mid \neg \mathsf{B} \quad (\oplus \in \{||, \&, \mathsf{shl}\})$

$\mathsf{A} \equiv \mathsf{B} \mid \mathsf{A} \otimes \mathsf{A} \quad (\otimes \in \{+, -\})$

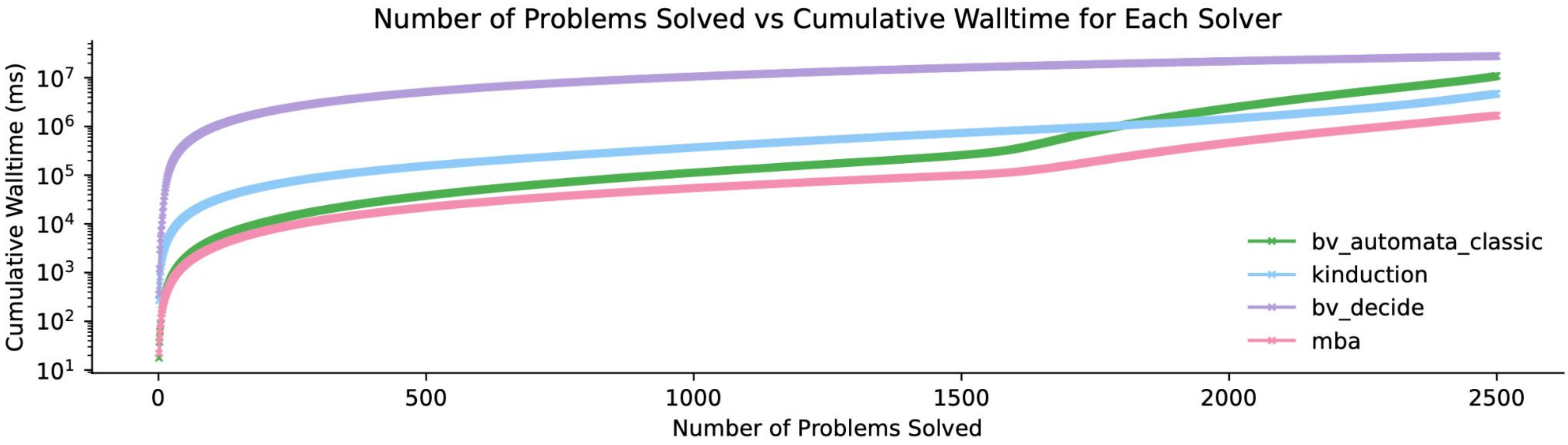
$\mathsf{P} \equiv (\mathsf{A} = 0).$

```
theorem add_eq_xor_and_w1 (x y : BitVec 1) :
    x + y - (x ^^^ y) - 2 * (x &&& y) = 0
```

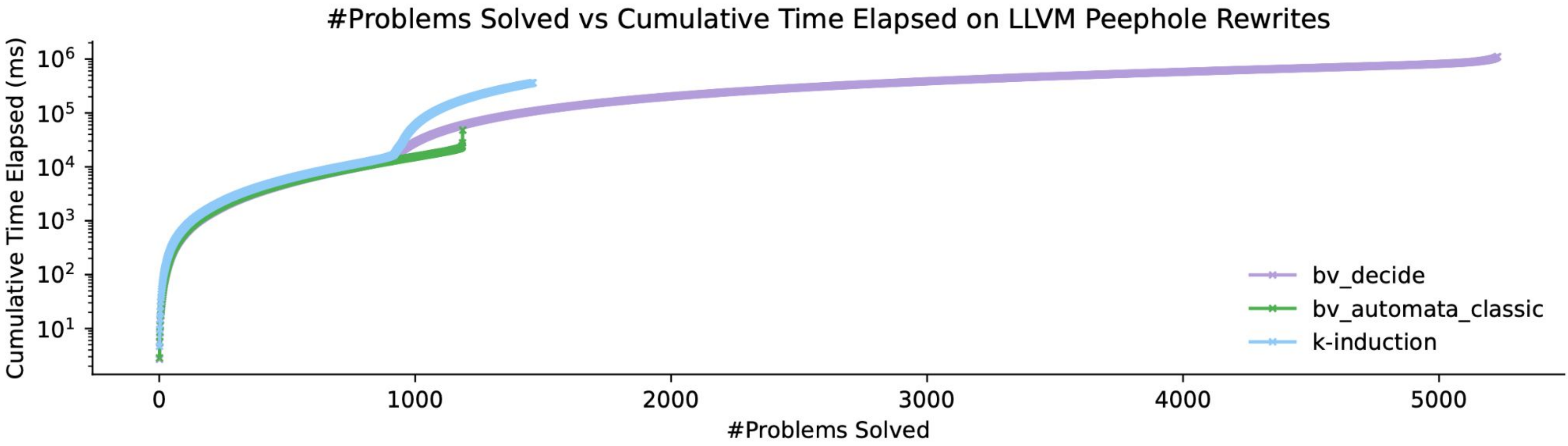
```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x1^y1)+(x0^y0)) - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1 - (x1^y1) -2 * (x1&y1) = 0
2(x0 + y0 - (x0^y0) -2 * (x0&y0) = 0
```

How Well Do These Algorithms Work?

Problems Solved v/s walltime on MBA-Blast

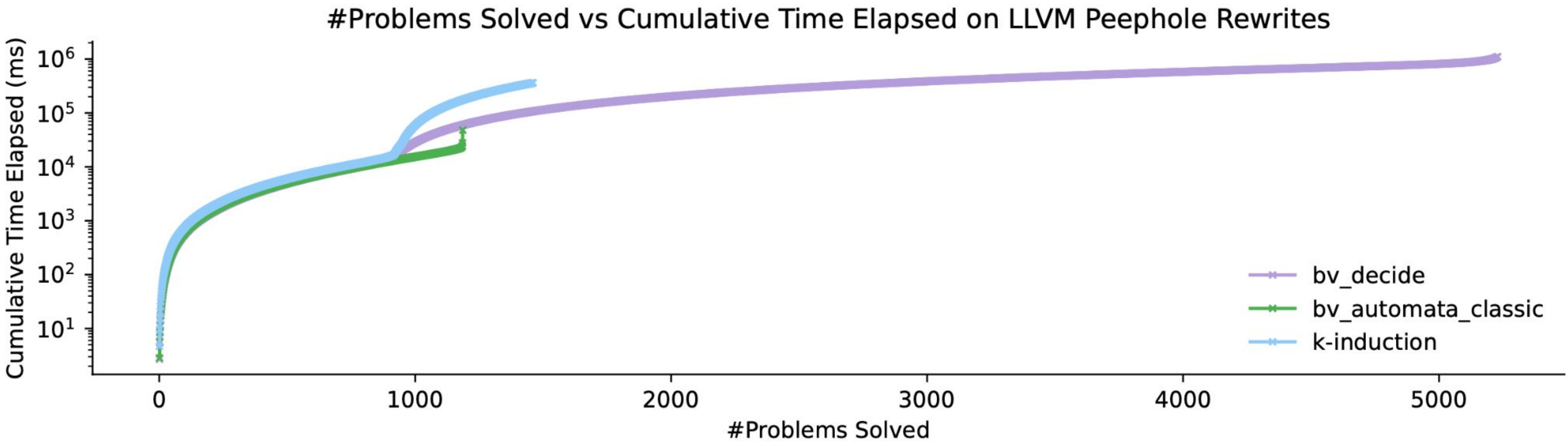


How Well Do These Algorithms Work?



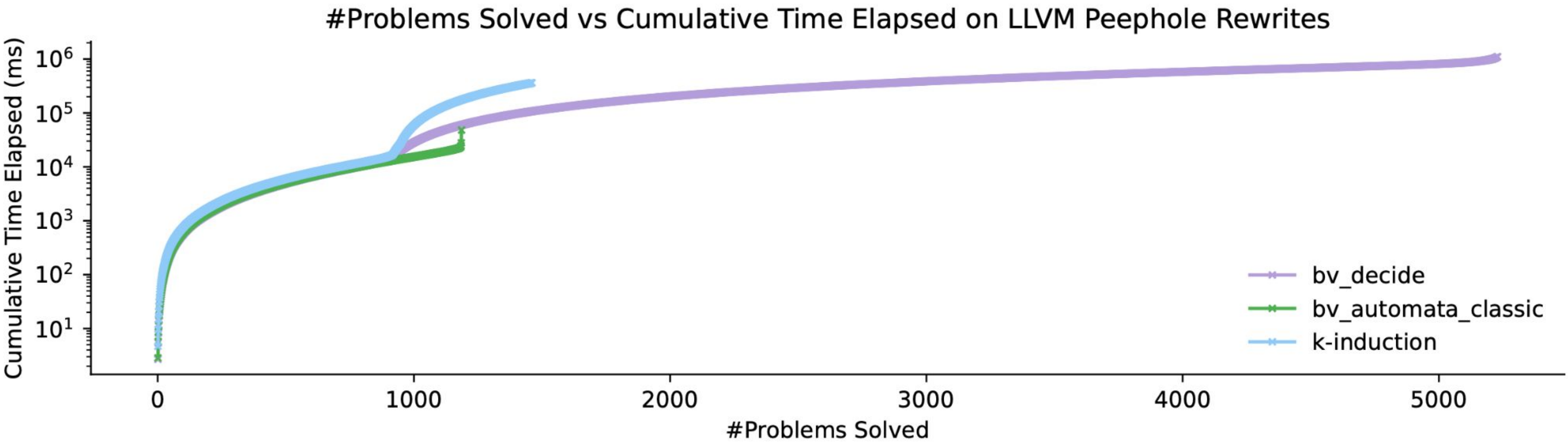
- Rewrites With Constants For Fixed Width: **~500** problems (7 → 2^w-1)

How Well Do These Algorithms Work?



- Rewrites With Constants For Fixed Width: **~500** problems (7 → 2^w-1)
- Rewrites With Multiple Widths: **~2000** problems (v, w, ...)

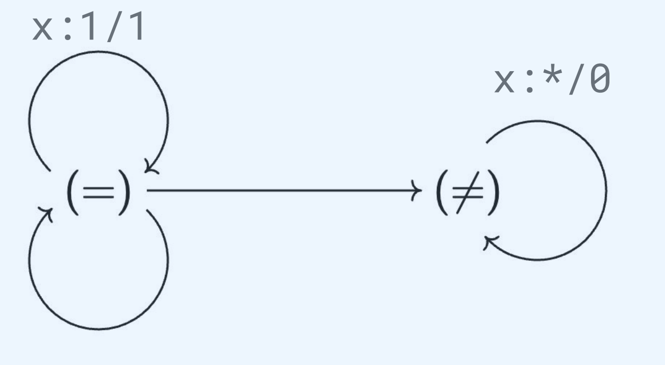
How Well Do These Algorithms Work?

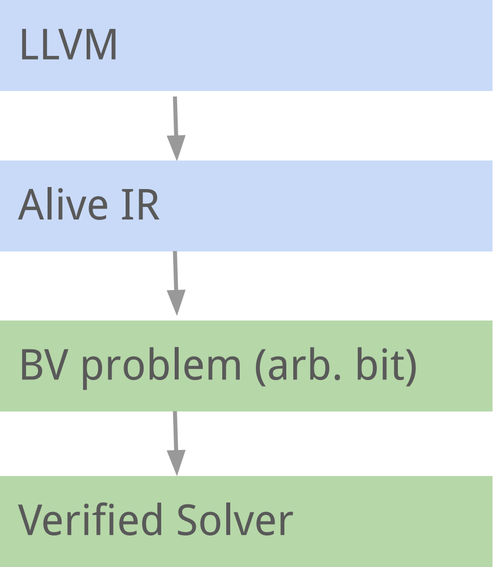


- Rewrites With Constants For Fixed Width: **~500** problems (7 → 2^w-1)
- Rewrites With Multiple Widths: **~2000** problems (v, w, ...)
- Rewrites With Mul / Div / ... : **~500** problems

Trust Your Rewrites With Arbitrary Width Solvers!

```
x1x0 + y1y0 - (x1x0 ^ y1y0) - 2 * (x1x0 & y1y0)
(2x1+x0) + (2y1+y0) - (2(x^y1)+(x0^y0))
   - 2 * (2(x1&y1)+(x0&y0))
2(x1 + y1 - (x1^y1) -2 * (x1&y1) = 0
2(x0 + y0 - (x0^y0) -2 * (x0&y0) = 0
```





LLVM

Alive IR

BV problem (arb. bit)

Verified Solver